Create and Explore a graph from Agda Definitions to analyse projects and speed up compilation

Dissertation

Julian Camilo Vidal Polania

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

Usually 100-300 words stating the salient points of the report. It should help your reader to decide whether the report is relevant to her or his interests.

I will mention:

- How I got the s-expressions and how they were parsed
- The uses of the graph and the queries you can make
- How much it can speed up compilation and how

Acknowledgements

Abbreviations

ACB	Apple Banana Carrot

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Background

Add more citations to articles and different sources
Add section about s-expressions

When you first encounter a massive code base, containing years of development from different people, all adding their code with their style and thought process, it can be overwhelming and difficult to parse through. On most codebases you can ignore the fundamental functions that underpin a given library, but in Agda it is quite useful to know the underlying definitions of the project.

Agda is a functional programming language that can be used as a proof assistant, following the propositions as types logic system [2]. Since proofs in Agda can be created almost entirely from scratch, defining the most fundamental types and operations means that projects will quickly balloon. While most of the time, it is possible to use higher-level definitions when developing proof, knowing how the types are fundamentally defined can be quite beneficial.

1.2 Motivation

Due to the size of these projects and how complex type definitions can be it becomes difficult to fully grasp what the relationships between all the definitions are. This creates the need for a tool that can analyze Agda projects and give the user an easy-to-use interface to query the relationships between definitions, to better understand the proofs. A tool that would allow the user to explore these relationships would be of great benefit.

Also, large Agda projects tend to deal with long compilation times. Agda is a proof assistant, during "compilation", it type checks the entire project which can take 5 minutes or more depending on the computer and project. Agda will keep track of what modules have already been compiled, so working on high-level modules isn't an issue as Agda will only type check files that have changed in the project and high-level modules don't have dependents that would need to be re-checked. In contrast, low-level files with many dependents, such as a module

defining the natural numbers will cause a significant portion of the project to re-compile. This effect will be particularly pronounced during refactoring where significant changes will be made to the bottom of the project, causing Agda to re-check the whole project making most of the time spent refactoring, spent waiting.

Agda type checks the project sequentially, going through the dependencies of each module, this is done to maintain safety as a module won't be compiled until its dependencies have. A tool that analyzes the relationship between modules and finding modules that are safe to compile in parallel could lead to significant time savings.

1.3 Problem Statement

Agda doesn't provide a built-in tool to extract the definitions from a project, but there is an s-expression extractor Agda version by Andrej Bauer [3] that will allow a better view of Agda's internal representation. S-expressions aren't well-suited data structure for the queries on definition relationships, mostly contain syntax information that can be discarded and each module has an s-expressions file. A data structure that lends itself well towards queries of the relationship between definitions, that only contains information about the definitions and stores all the definitions from a project, such as a graph. A tool that could explore the dependency graph of the definitions, would allow for a faster understanding of a large code base.

While Agda doesn't have a tool to extract definitions from projects, it does have a tool to create a dependency graph from the modules. For compilation time improvement, only the modules are of interest as such an effective representation is already available. The challenge is what is the most effective way to traverse a dependency graph, concurrently, while maintaining safety (i.e. don't compile modules and their dependencies at the same time).

1.4 Objectives

The objective of this project is to create a dependency graph of definitions from an Agda project by using the s-expression representation of modules using the s-expression extractor [3]. Then create build a (Command Line Interface) CLI will create the dependency graph from any Agda project and give the user access to commands that query the dependency graph. These queries will let the user explore the graph, to gain insight into the Agda project. As there aren't currently any tools that allow you to query the definitions of an Agda project.

The objective is also to build another CLI tool that will create the module dependency graph from an Agda project and generate the makefile with the optimal order that modules should be compiled to reduce compile time.

1.5 Project Structure

First, there will be an explanation of what s-expressions are and how they are being read. These s-expressions use tags to name different symbols in a module,

only the tags that contain definitions are important. Then, show how the s-expressions are structured, the way that definitions can be found, and their dependencies. Using the information gained for the s-expressions, they will be turned into a graph that can be easily queried for relationships. Some example queries will be demonstrated, to underline their utility and implementation.

Secondly, by analyzing different algorithms for traversing a module dependency graph there will be an exploration of how parallelism can improve the compile time of an Agda project. Results from different algorithms and projects will be shown.

CHAPTER 2

Literature Review

There are many ways to represent code using graphs, each with their own utility and purpose. There are Abstract Syntax Trees (AST), Control Flow Graphs (CFG), Data Flow Graph (DFG), Program Dependence Graph (PDG), and Code Property Graph (CPG). These representations encode all the behavior and properties of a project, they can have many uses such as for code vulnerability detection [12]. These representations also allows for static analysis, where code isn't run but the structure is analyzed for software validation [11]. For this project, these graphs encode far more information that is required, as for the dependency graph only the relationship between definitions is of value.

Add literature about code decay

Add section explaining dependency graph

2.1 Related works

The Language Server Protocol (LSP) is used by IDEs, such as Visual Studio Code and IntelliJ, to provide features like goto definition and goto references. The LSP sits between the code editor and the language server, it is the language server that analyzes the code structure to support the features [13]. Language server has the functionality to implement this project, but, they are meant to be used while editing code, and the language server isn't made to be used with custom queries. Examples of LSPs are Jedi [17] and Agda's language server [1], while they contain the functionality needed they are only meant to work on the current open file, not a whole project.

Ctags [20] is a tool that indexes all the symbols in a project, this is helpful to get all the definitions from an Agda project. But it doesn't capture the relationship between the symbols it finds.

Graph Buddy is an interactive code dependency browsing and visualization tool [6]. It takes large Java codebases and turns them into Semantic Code Graphs (SCG), and creates a visualization of this graph. This graph shows dependencies between modules, classes, and methods which helps the developer better understand the project and tackle the pervasive issue of code decay

[4]. Graph Buddy is integrated into an IDE as a plugin, where the user can seamlessly explore the visualization.

There is also a tool [8] that visualizes the call graph of a coding project, this allows for better developer experience. It works in real-time, while the user is editing the code the graph will automatically update to show the changes.

2.2 Parallel Compilation

Add section explaining DAGs

Type-checking is a computationally expensive task that hinders the workflow of a developer, this has led to work to parallelize type-checking algorithms or make them incremental. Parallel type-checking aims to type-check different parts of a project at the same time, while incremental type-checking aims to allow the developer to type-check a small change in the project without having to type-check the whole project again.

An example is the work by Newton et al [16] which seeks to parallelize type-checking with Haskell. Also, the work by Zwaan et al. [22] using scope graphs for incremental type-checking.

However, this project doesn't aim to optimize the type-checking algorithm themselves, rather, find independent modules that can be type checked together. This aligns more closely with the following paper that explores reducing FPGA compile time by changing from a monolithic compilation style to compiling separate blocks in parallel [21].

Add section explaining scheduling problem

Fundamentally the compilation problem in this project is a scheduling problem, which is NP-complete[15]. This means that there many algorithms that attempt to tackle this problem by following different assumptions as shown by Yves Robert [15].

2.3 MLFMF: Data Sets for Machine Learning for Mathematical Formalization

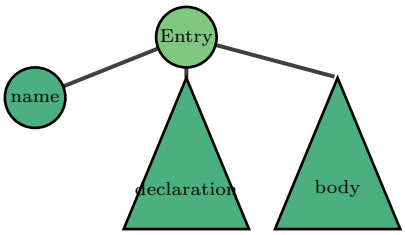
Add section explaining s-expressions

MLFMF is a collection of data sets used to benchmarking systems that help mathematicians find which theorems are relevant when proving a new theorem. The data sets are created from large libraries of formalized mathematics in Agda or Lean. The data sets represent each library as a network and as a list of s-expressions. The data sets include libraries such as Agda-unimath and TypeTopology. The collection is a solid base for investigating machine learning methods to mathematics. The data set methodology to extract s-expressions can be used with other libraries to continue expanding the 250000 entries in the data sets.

The method to extract s-expressions from Agda libraries is by extending the Agda backend. The backend has access to important information about a project's theorems and its relationships with other theorems. Andrej Bauer used that backend and turned the information into easy-to-parse s-expressions. The Figure 2.1a is an example of the :entry s-expression, each :entry tag marks where a theorem is defined, and it contains the name of the theorem, its type, and the definitions it needs to be defined. Mind that `(...)` are more s-expressions that were replaced for readability. A more general structure to this tag can be seen in Figure 2.1b where the three parts are shown as subtrees and a node.

```
(:entry
  (:name ℕ)
  (:type (...))
  (:data
    (...)
    (:name ℕ.zero)
    (:name ℕ.suc)
  )
)
```

(a) Example s-expression from MLFMF Figure 1(b) [5]



(b) Graph representing s-expression entry tag containing a name, declaration and body from MLFMF Figure 3 [5]

The :entry tag is of particular interest to this project, as Agda by itself doesn't provide a convenient method to find all the definitions of an Agda project along with its declaration. This extractor packages the important information into a format that is easy to parse into a graph. Andrej Bauer made the s-expression extractor open-source in a GitHub repository [3].

This paper also describes a graph that can be generated from these s-expressions, this graph contains all the information about each theorem and what definitions it uses and in which way. While this graph could be used queried by the user, it contains a massive amount of information that is unnecessary for the user to know the query.

- What the paper is about
- How the paper uses s-expressions
- What the s-expressions say
- Add a graphic of the s-expression structure from the arxiv paper, well cited

2.4 Conclusion

Many tools analyze and visualize the overall structure of programming projects. Allow for static code analysis, where software can be validated and developers can have an easier time exploring a project. However, most of the tools are not easy to query by the user, are meant to be used while editing a file and are reserved for more popular languages like Java. A tool that can read an entire Agda project and gives the user access to the underlying graph is still missing.

The MLFMF paper [5] describes a methodology to extract the definitions and dependencies of an Agda project into s-expression. Which are easier to parse than Agda source code. The also converts this into a graph that contains all the details of the Agda source code. Which is more detailed than a user who wants to query that information needs.

Slow compilers are a common problem, hurting developer experience. Due to the monolithic nature of compilers, parallelization becomes a route to follow when optimizing compilation time. While parallelization can be applied to the type-checking algorithm itself, this project looks to type-check modules in parallel while the type-checking algorithm remains the same. This problem is

closer to a scheduling problem, where the goal is to find the optimal way to assign tasks to multiple machines to reduce completion time. No tool in Agda attempts to apply a scheduling algorithm to the type-checking of modules, which could lead to significant speed-ups.

CHAPTER 3

Legal, Social, Ethical and Professional Issues

The project uses several open-source libraries, such as NetworkX and sexpdata. It is important to comply with the licenses of these libraries, such as MIT and BSD to avoid legal issues. Without compliance, this tool can't be made publicly available. This project does not handle sensitive user data, any future additions that might involve user data must comply with data protection regulations.

The project has the potential for a positive impact on developer productivity by giving tools to understand large codebases quickly and reducing downtime during refactoring. This tool needs to be accessible and inclusive to a wide range of users. Since the tool runs on a terminal the user can choose the interface that suits them best. It is also important to provide documentation and helpful error messages to help users unfamiliar with Python to use this tool.

Transparency and accountability are critical for a project that will analyze the personal and professional projects of developers. The tool must give accurate results when querying definitions and creating compilation order. Misleading output can introduce errors during development which is unethical and counterproductive.

This project must follow the professional standards set by the BCS Code of Conduct. These standards ensure that software is reliable, secure, and ethical. The tool will evolve and it is critical for it to be well-tested, to avoid introducing errors.

CHAPTER 4

System Requirements

The CLI tools should be easy to install by any developer, the developer shouldn't need any extra knowledge to use the tool. The interface must be intuitive and there should be little friction between the development workflow and the use of these tools. Also, the tools must work in a variety of environments without issues.

4.1 Agda Tree

The Agda Tree CLI tool is an application that will run on the terminal. It will extract and save the definition dependency graph from an Agda project, then it will give the user commands to query and explore that graph. This tool should work with any Agda project and the user should be able to install the tool without any extra dependencies.

4.1.1 Functional Requirements

The functional requirements are the features that the tool must implement to be usable and meet the expectations of users. For the tool to be easy to use, it must be able to automatically extract the dependency graph from any Agda project. The user should be able to easily query the dependency graphs and the output of the queries should be intuitive to understand.

Table 4.1: Agda Tree Functional Requirements

ID	Name	Description
1	Definition Dependency Graph Extraction	The tool must parse Agda projects and construct a definition dependency graph
2	Querying the Definition Graph	The CLI must allow the users to query the dependency graph to retrieve important information. (See table 4.2 for queries)
3	Command-Line Interface	The tool must provide a user-friendly CLI with commands for querying the dependency graph
4	Input Validation	The CLI must validate user input and provide clear error messages for invalid inputs
5	Integration with Agda Projects	Agda projects are structured differently, this tool must support all valid structures
6	Output Generation	The CLI must display the query results in a readable format that follows the style of other Unix CLI tools
7	Module Dependency Graph Extraction	The tool must parse Agda projects and construct a module dependency graph
8	Querying the Module Graph	The CLI must allow the users to query the dependency graph to retrieve important information. (See table 4.3 for queries)

Add queries that weren't on mastodon what are on the cli

Martin Escardo asked in Mathstodon [9] for all the possible queries that this tool should implement. This tool will allow for the querying of both the definition and module dependency graphs, the queries that can be made on the definition graph are the following:

Table 4.2: Agda Tree Definition Queries

ID	Name	Description
1	Dependencies	Get the dependencies of a definition and what definitions it uses both directly and indirectly
2	Dependents	Get the dependents of a definition, where the definition is used both directly and indirectly
3	Leafs	Gets the leaves of the dependency graph, which would be the definitions that have no dependencies
4	Module Dependencies	Gets the modules that a definition uses both directly and indirectly
5	Module Dependants	Gets the modules that use a definition both directly and indirectly
6	Path to Leaf	The longest path from a definition to any leaf
7	Module Path to Leaf	The longest path from a definition to any leaf but only following the modules of the path
8	Roots	The definitions with no dependents, meaning they aren't used anywhere
9	Definition Type	The definitions used for the type of the definition
10	Use count	Counts how many times a definition is used
11	Cycles	Returns the cycles in the graph
12	Save Tree	Saves the tree into a dot file
13	Path Between	Finds the longest path between two definitions

The queries that can be made on the module are mostly the same the module graph is a directed acyclic graph (DAG) giving it special properties. The queries are the following:

Table 4.3: Agda Tree Module Queries

ID	Name	Description
1	Dependencies	Get the dependencies of a module and what modules it uses both directly and indirectly
2	Dependents	Get the dependents of a module, where the module is used both directly and indirectly
3	Leafs	Gets the leaves of the dependency graph, which would be the modules that have no dependencies
4	Path to Leaf	The longest path from a module to any leaf
5	Roots	The modules with no dependents, meaning they aren't used anywhere
6	Use count	Counts how many times a module is used
7	Level Sort	Returns a list of modules sorted by how far away it is from a leaf
8	Path Between	Finds the longest path between two modules
9	Topological Sort	Returns a list of modules sorted topologically

4.1.2 Non-Functional Requirements

Since Agda Tree is meant to be a tool that slots into the workflow of developers, it must be easy to use and performant. The tool is plug-and-play, working on a variety of projects regardless of size. To not disrupt the developer, the queries should respond quickly. Agda is used mainly with Unix-based systems, so this tool must be compatible with macOS and popular Linux distributions.

Table 4.4: Agda Tree Non-Functional Requirements

ID	Name	Description
1	Extraction Performance	The tool must extract the dependency graph in 10 minutes depending on the size of the project
2	Query Performance	The tool must be able to respond to a query in under 2 seconds
3	Scalability	The tool should allow for fast querying of large projects
4	Usability	The tool must be easy to use, with intuitive commands, clear documentation, and meaningful error messages.
5	Compatibility	The tool should work on macOS and Linux
6	Reliability	The tool should handle bad inputs gracefully
7	Maintainability	The codebase should be well documented and well-structured to allow for new queries
8	Testability	The queries should be tested to ensure the correct output is given

4.2 Agda Comp

The Agda Comp CLI too, is an application that will run on the terminal. It will extract the module dependency graph from an Agda project and produce the order in which the modules should be type-checked.

4.2.1 Functional Requirements

The functional requirements for Agda comp are the features that the users need, to compile their projects with minimal hassle. This tool must work mostly automatically, the user only needs to input what they want to compile, and the tool will determine all the information it needs from there. The tool will create index files and a make file that will compile the Agda Project based on the selected strategy. This compilation must be safe and correct, it must compile all necessary modules and it shouldn't compile two modules concurrently.

Table 4.5: Agda Comp Functional Requirements

ID	Name	Description
1	Module Dependency Graph Parser	The tool must parse Agda's dot file module dependency graph
2	Compilation Strategies	The users must be able to select which compilation strategy to use
3	Compilation Customization	The users must be able to customize parameters of the compilation strategy (i.e. amount of cores used)
4	Compilation	The tool must create index files and a make file that will safely compile all modules
5	Command-Line Interface	The tool must provide a user-friendly CLI with commands for querying the dependency graph
6	Input Validation	The CLI must validate user input and provide clear error messages for invalid inputs
7	Integration with Agda Projects	Agda projects are structured differently, this tool must support all valid structures
8	Speed Up Compilation	The tool must compile Agda projects faster than a normal compilation.

4.3 Non-Functional Requirements

Agda Comp is meant to be an additional tool that slots in next to Agda seamlessly, so users who already have Agda mustn't need extra setup. The tool should be easy to use and work with any Agda project. Agda is not developed for Windows, so the main focus is for the tool to work in a Linux or macOS environment.

Table 4.6: Agda Tree Non-Functional Requirements

ID	Name	Description
3	Scalability	The tool should allow for compilation of large projects
4	Usability	The tool must be easy to use, with intuitive commands, clear documentation, and meaningful error messages.
5	Compatibility	The tool should work on macOS and Linux
7	Maintainability	The codebase should be well documented and well-structured to allow for new queries
8	Testability	The compilation strategies should be tested to ensure the correctness and safety

4.4 Conclusion

The functional requirements for Agda Tree ensure that it can be widely used with little setup with outputs that are useful to the user. It is important to make certain that all the queries an Agda developer would need are included, along with a simple way to use the output of the tool. The non-functional requirements define how the Agda Tree must behave, for it to slot into a developer's workflow.

The functional requirements for Agda Comp show the features needed for the tool to be used, the tool must work with any Agda project and give the user a choice on how to run their compilation given some parameters. The non-functional requirements define the behavior of Agda Comp, it must work with any Agda project regardless of size and the compilation strategies should be correct and safe. That is the compilation strategies should compile every module necessary and do so without compiling two modules at the same time which could cause concurrency issues. Since the time it takes to compile varies on the project size, performance measures aren't going to be made.

CHAPTER 5

Design

Add an introduction of what you want to do for each chapter

The design of tools will describe the different systems needed to extract the dependency graphs and analyse them. Agda Tree analyses two dependencies graphs, one for modules and another for definitions. The module dependency graph can be generated with Agda but for the definition dependency graph will have to be created manually and needs to be designed.

Agda Comp is a simpler tool, as the complexity comes from testing the different compilation strategies. It must create the module dependency graph, apply the given strategy using the parameters provided by the user, and run the compilation order.

5.1 Agda Tree

Agda Tree is a command line interface that allows the user to interact with the module and definition graph. The first command that has to be designed is how the CLI will create both dependency graphs. Figure 5.1 shows how the user will interact with the CLI to create the graphs. The user provides the Agda file that they want to analyze, normally this would be the entire project. The "Everything Index File" is the file that imports all the modules in a project, this allows the user to analyze the entire project. This is standard in Agda, as this is the only way to get a file that type-checks the whole project. Depending on the Agda project, this file will be generated automatically or by the project maintainers.

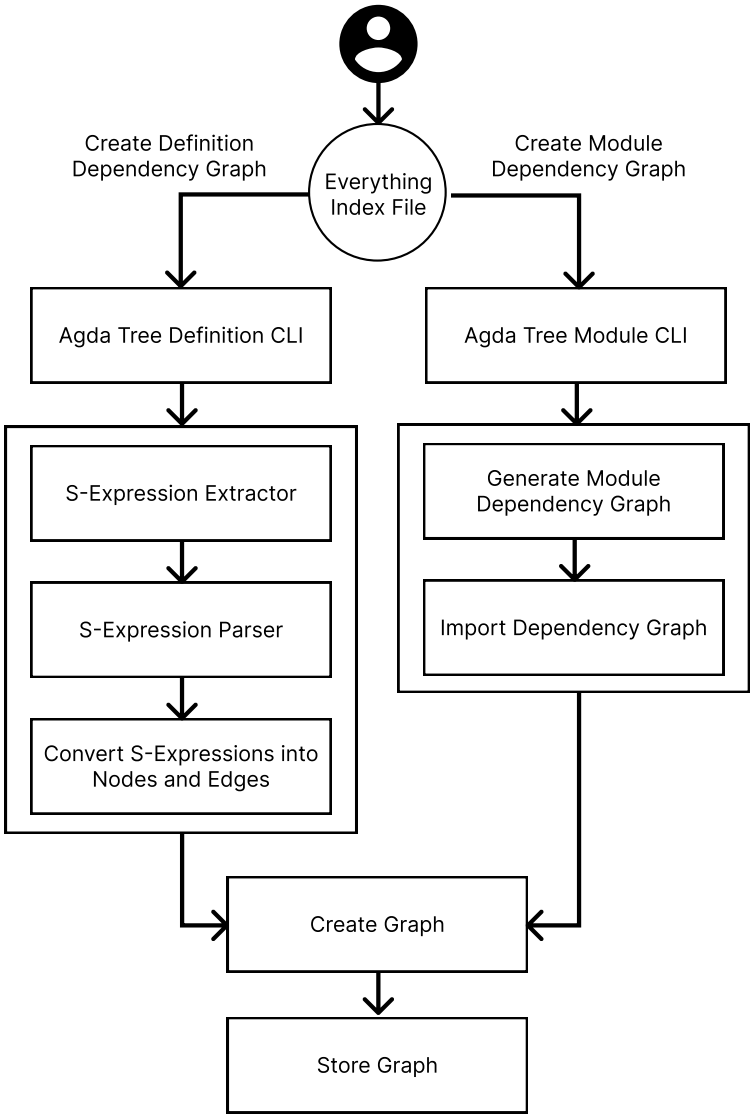


Figure 5.1: Agda Create Tree Diagram

Figure 5.2 is a Class Diagram, showing the classes and methods needed to operate the command line. The program starts with the command line entry point, that will parse the input by the user and delegate the running of the commands to the respective dependency graph. The command line stores the default path to the dependency graph, the definition and module dependency graph will have their own commands that can be run on them. These queries can be found in Table **??** for the definition graph and in Table **??** for the module graph for the module graph.

For the definition graph to be created, it depends on the s-expression extractor and parser. They will read the Agda projects and convert the data found into a graph.

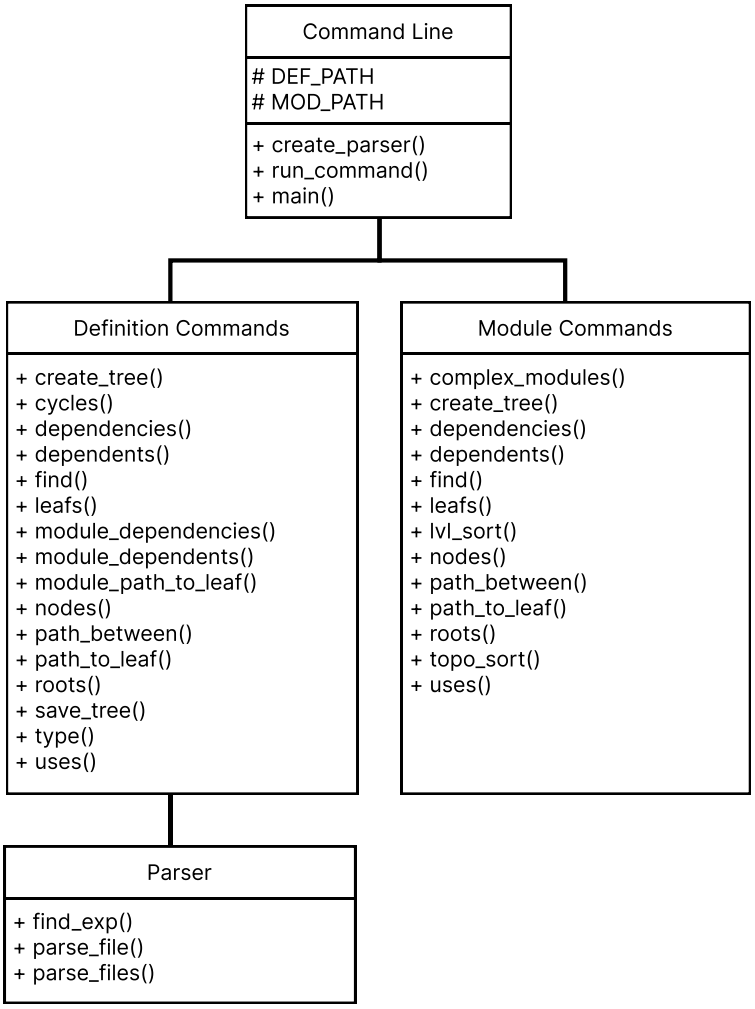


Figure 5.2: Agda Tree Class Diagram

Figure 5.3 shows how the CLI splits into two, depending on what dependency graph is being queried. When a user makes a query, they will select the dependency graph and the respective methods will perform that query. The output will be displayed in stdoutput, this makes sure that the output can then be used with other commands like piping and xargs.

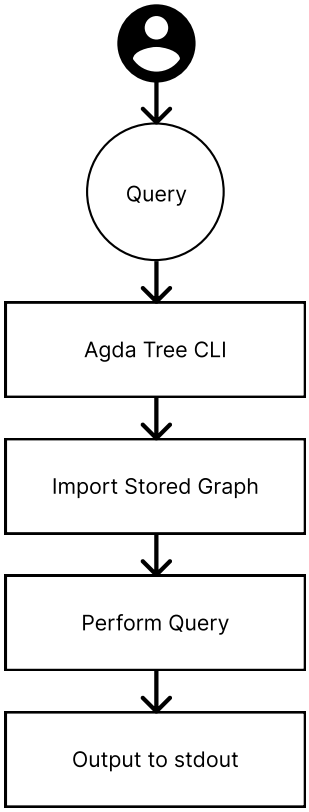


Figure 5.3: Agda Tree Query Diagram

5.2 Agda Comp

Figure 5.4 demonstrates how the user will interact with the Agda Tool. The user provides the module that will be compiled, next to some parameters defining the amount of cores used in the parallelization and what compilation strategy to use.

Add more stuff about Agda Comp, I don't know what
What are the algorithms that are going to be sued for Agda comp

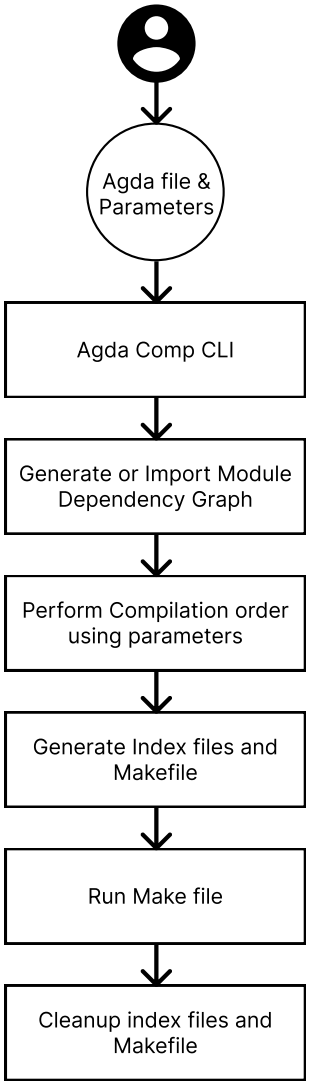
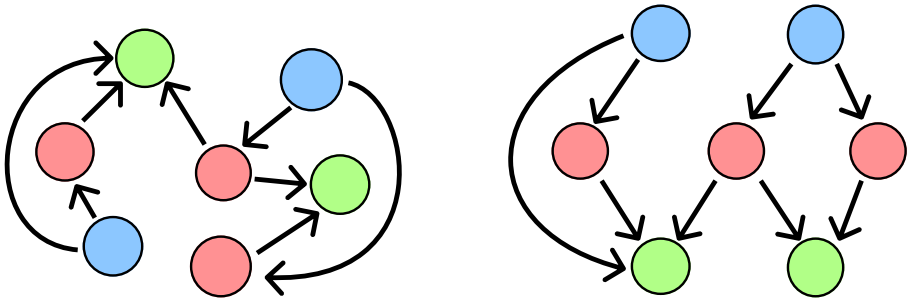


Figure 5.4: Agda Comp Diagram

5.2.1 Level Strategy

The level strategy sorts the modules into levels, where leaf nodes with no children are at level 0. Level 1 contains all the modules which only depend on modules at level 0. Level n contains all the modules that depend on levels $n-1$ or below. In other words, the level of a module is its maximum distance from a leaf module. This algorithm can be visualized with the Figures 5.5.



(a) Example dependency graph

(b) Example dependency graph sorted by levels where green means level 0, red level 1 and blue level 2

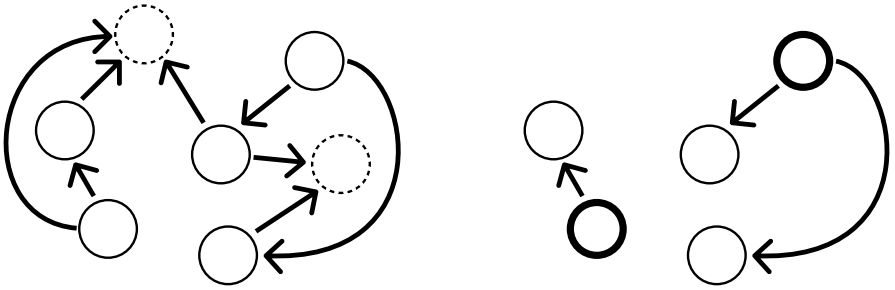
Figure 5.5

This sorting has the property such that each level only depends on the levels below it, meaning if the below modules were already compiled the modules at the current level could all be compiled in parallel. The level strategy is to sort the modules into levels, then compile each level linearly and the modules at each level are compiled concurrently. This strategy also compiles every module in the graph as the sorting keeps all the modules in the dependency graph and the level strategy compiles every level. Therefore, this algorithm is both safe and correct.

The advantage of this solution is that it can be quickly generated recursively, where each module will take the maximum of the recursive call to its children and add one the maximum of the children's level. The disadvantage is that if a level has a small amount of modules there isn't a significant opportunity to parallelize the type checking. Also, There could be two modules that each depend on 20 distinct modules that could be compiled in parallel that could lead to massive savings. Instead, this method would compile the 40 combine dependencies in previous levels, then compile the two modules in parallel. This becomes significant during implementation **??**, as there is an overhead to parallelization which is exacerbated when compiling a small collection of modules.

5.2.2 Level Disjoint Strategy

The level disjoint strategy aims to target the weakness of the level sort strategy. It aims to find the largest modules that can be compiled in parallel, the largest module being one with many dependencies. If such modules are found, then they are compiled in parallel, otherwise the leaf modules are compiled. Note that once a module is compiled, it is no longer a dependency on future modules. This process is visualized on Figures 5.6.



(a) Example dependency graph

(b) Example dependency graph sorted where the leafs were removed, creating two modules that can be compiled in parallel.

Figure 5.6

This ensures that at each step the amount of module to compile is reduced, until all modules are compiled. It also ensures that if no distinct modules, modules that don't share dependencies, are found then the conflicting dependencies might be the leaf that is being compiled. The strategy only compiles two modules in parallel if their dependencies are disjoint, if such modules aren't found then the leafs are compiles which don't have any dependencies. Therefore, this strategy is both safe and correct, as it doesn't compile conflicting modules and compiles all modules in a project.

The advantage of this approach is that it better manages the overhead of parallelization, instead of compiling a couple of individual modules at a time it can compile modules with multiple dependencies such that Agda doesn't have to loading interface files as often as explained in the implementation Sub-Section **??**. The disadvantage is that finding these disjoint modules is difficult, projects have hundreds of modules and finding multiple modules that don't share dependency can't be done through brute force. Each Agda project also has different structures with Agda-unimath having smaller independent modules while TypeTopology has bigger dependent modules that make finding distinct module more difficult. A greedy approach has to be taken, which doesn't guarantee the optimal solution.

5.3 Conclusion

The diagram in figure 5.2 show how the structure of the program. Figures 5.1 and 5.3 also shows how the dependency graphs will be created and how the user will interact with CLI. This structure will be important as it ensures that the CLI can handle all the requirements and gives the project a solid foundation to refer back to.

The functionality of Agda Comp is modelled in Figure 5.4, this structure allows the user to select what compilation strategy to use and how many cores the compilation can use. Agda Comp is meant to allow the user to easily apply the compilation strategies which are modelled with Figures that are going to be explored in chapter 6.

CHAPTER 6

Implementation

The implementation of Agda Tree must create a dependency definition graph from an Agda project and allow the user to query this graph through a CLI. First, a subsystem that turns Agda projects into a dependency graph needs to be implemented. Then, the CLI that will let the user query the graph is implemented. These subsystems must meet the functional requirements described in 4.1 and the non-functional requirements described in table 4.4. The CLI must also implement all the queries described in table 4.2 for the definition graph and the queries described in table 4.3 for the module graph.

6.1 Agda Tree

The definition graph is the most important and difficult graph to generate. It contains all the detailed information needed by the developer to decipher the structure of the definitions and their relationships. Agda can already create the module dependency graph, but there is no native feature that can create a definition dependency graph.

6.1.1 Building the definition tree

At first, a possible option was using the HTML files that Agda can create. These HTML files display and format the code with colours and links. This is vital because the HTML files would style the text differently depending on if the text was a definition, keyword, type, or operator. Also, all the used definitions were hyperlinks that connected back to the module that defined them. With a method to find the definitions in a file given the styling and finding the dependencies of that definition given the hyperlinks, a graph could be created. The main issue was parsing the HTML files, finding which keywords fell into which definition was quite difficult, and using an Agda parser might have been necessary.

The solution is not to use the HTML files but to use s-expressions. The s-expressions are the same way the HTML files are with the same information,

except s-expressions are easier to parse. However, Agda doesn't natively generate s-expressions but Andrej Bauer, Matej Petković, Ljupčo Todorovski in their paper "MLFMF: Data Sets for Machine Learning for Mathematical Formalization" [5] created an s-expression extractor. The s-expression extractor is in the Agda backend, and it will convert Agda files into s-expressions [3].

S-expression extractor

S-expressions are a notation that is used in Lisp programming languages, it represents programs and data as tree-like data structures [7]. The grammar for the s-expressions varies, but for this case the s-expression are of the form: (:tag sexp-1 sexp-2 ... sexp-n). Where sexp-n can be a number, a string or another s-expression and the tag is a keyword that describes the content of the s-expression. The "MLFMF" paper describes in more detail the structure of the s-expressions with respect to Agda [5].

Here is a brief summary of the relevant s-expressions that are needed for the implementation:

Table 6.1: Relevant S-expressions

sexp	Description
(:module module-name entries...)	The root tag that holds the whole module, module-name is the name of the modules and entries are the definitions in the file
(:module-name name)	The module name
(:entries name type body)	The definition, it includes its name, type and the body of the definition
(:name name)	The name of a definition, this name can appear as the name of an :entry tag, within the :type or :body tag
(:type type)	The description of the type of the parent definition
(:body body)	The body of a definition

Note that the s-expression extractor is a special version of Agda with an extended back-end, which means the user has to compile this Agda version and add it to their path. This will be handled by the CLI described section **??** to make sure the usability non-functional requirement is met from Table 4.4.

S-expression parser

The :body and :type tags contain other tags like :apply, :sort, :max, etc. That describe the definition in full detail, but this information is not needed. The information needed is the definitions in a project and the dependencies of those definitions which means mainly the :name and :entry tags are of interest.

The s-expressions are compiled into a directory, this project is implemented in Python so the library sexpdata will be used to load the raw s-expression files into Python lists. These lists will be analysed recursively, finding the relevant

Should I add the figure in MLFML that describes the s-expressions

tags. The strategy is to find all the :entry tags in the file, each :entry tag represents a definition as described in Table 6.1. For each :entry tag, find all the :name tags contained inside. With this information create a dictionary where the key is the definition name and the value will be a list of the :name tags. Since :entry tags describe each definition and the :name tags describe what definition is being used, the resulting dictionary will contain all the definitions along with their dependencies. To find the necessary tags a find_exp function is implemented that recursively finds all occurrences of a given tag within an s-expression.

The s-expression extractor writes an s-expression file for each module, so the mentioned dictionary is created for each file and combined into one big dictionary that has all the definitions from the entire project. The same process will be repeated but instead of looking for :name tags inside the entire :entry tag, find all the :name tags in the :type tag. Store this in a dictionary with the definition as a key and the :names found in :tag as the value. This provides the information about the type of the definition.

This parsing procedure is done in parallel, where each file is parsed in its own thread and the dictionary of all the parsed files is combined by adding all the key value pairs into a bigger dictionary.

Building definition graph

The dictionary with definitions as keys and their dependencies as values, already forms the definition dependency graph. Which can be imported into the NetowrkX Python library. This library efficiently creates and queries graphs, it contains many useful features and performance benefits. NetworkX is a widely used tool, so a user can easily become familiar with its use and implement their own queries.

The graph is first initialized with the command nx.Digraph() then all the definitions are added as nodes with the command graph.add_nodes_from, where the key values of the dictionary from the parsing are given. Lastly, the edges are added with the command graph.add_edges_from where an array is passed in containing tuples where each tuple has the key and its dependency. Note that when creating the edge the definition is the first, and it's the decency is second, since this is a directed graph this will cause the direction of the edge to be from the definition to its dependency. Once generated it will be 'pickled', a Python library that serializes Python objects, the trees will be serialized and stored for future use.

When parsing the s-expression files, the definitions will have their names with its module path along with an identifier number. Due to the way the s-expressions are extracted and the way Agda sometimes works two distinct definitions will have the same name, so an identifier is given to distinguish them. But this can be cumbersome for the user to deal with, so before the graph is created the tool will attempt to remove this number unless it causes an ambiguity.

6.1.2 Building module tree

Agda has a built-in feature that creates a DOT file containing all the modules in an Agda project, including the relationship with its dependency. This com-

mand is agda –dependency-graph=[PATH] [Index File PATH]. The DOT language describes how to create nodes and edges in a Graphviz graph. This is a standard format for graphs, NetworkX already has an extension that uses pydot, a Python library to read, write and create DOT files. The extension can import DOT files into NetworkX and write them back to DOT files, so once Agda generates the DOT file it can simply be imported into NetworkX where it can be used by the tool. This graph will then be 'pickled' and stored for future use.

6.1.3 Commands

Most queries are defined similarly between the definition graph and the module graph but the module graph being acyclic means that it has different properties. These queries can be explored in further detail in its repository [14].

Create Tree

The create tree command will create the dependency graph for either definitions or modules. It will perform the parsing and extraction automatically and save the graph to the home directory or the path of option -o.

```
agda_tree definition create_tree "source/AllModulesIndex.lagda"
    -o=~/.agda_tree/def_tree.pickle
```

Add an example of the command and a man page describing its options

Nodes

The nodes query gets all the nodes in the graph, it returns a list with all the definition names. For the -c option it will return the amount of nodes in the graph.

```
agda_tree definition nodes
agda_tree definition nodes -c
```

Find

The find query gets all the names that match the pattern. The user provides a regex pattern to match on, and the query returns all the names that match. There is a -name option, if true then it will match the pattern to the name of the definition if it is not set then it matches on the whole name including the modules the definition it is stored in.

```
agda_tree definition find "\_\+\_"
agda_tree definition find "\_\+\_" -name
```

Dependencies

The dependencies query gets all the dependencies of a definition. This means what theorems do the definition need to be defined, either due to their type or what it used to be proved true. Since this is a directed dependency graph and the

definition's edges point towards its dependencies, the dependencies are the children of the definition. NetworkX provides a method for this: graph.successors(definition).

This query must also allow for finding the indirect dependencies of a definition, not only the direct dependencies. NetworkX provides a method for this as well: nx.descendants(graph, definition). This will find the dependencies and the dependencies' dependencies recursively.

```
agda_tree definition dependencies
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
agda_tree definition dependencies -i
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Dependents

The dependents query gets the dependents of a definition. The dependents would be the theorems that use this definition either in their type or their body. In this dependency graph, the dependants' edges point towards the definition, so the parents of a definition are its dependants. NetworkX provides a method to get the parents and the parent's parent recursively which are graph.predeccessor(definition) and nx.ancestors(graph, definition) respectively.

```
agda_tree definition dependents
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
agda_tree definition dependents -i
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Leafs

The leafs query gets the definitions that have no dependencies, meaning no children. This can be found by looping through each definition and checking how many outward edges it has with the command graph.out_degree(node), if none they are a leaf.

```
agda_tree definition leafs
```

Module Dependencies

This query is exclusive to the definition graph. The module dependencies query will take the output of the dependencies query and only keep the module the the definition was in. Although, this will cause repetitions from multiple definitions in the same module, so they are added to a set to remove repetitions.

```
agda_tree definition module_dependencies
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
agda_tree definition module_dependencies -i
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Module Dependants

This query is exclusive to the definition graph. The module dependants query will take the output of the dependents query and only keep the modules of dependents then add them to a set to remove repetition.

```
agda_tree definition module_dependents -i
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Path To Leaf

The path to leaf query finds the longest path from a definition to a leaf. The definition query is used to get the leaves of the graph, then NetworkX has a method nx.all_simple_paths(graph, definition, leaves) which finds all the simple paths between two nodes. Simple paths are paths where no vertex is repeated. Once all the simple paths are found, they are measured for length and the biggest one is returned.

The definition dependency graph isn't acyclic while the module dependency graph is, this will cause a difference in performance of this command. The definition graph also contains significantly more nodes than the modules graph, so the amount of paths grows quickly.

```
agda_tree definition path_to_leaf
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Roots

The roots query gets the definitions that aren't used by any other theorem, it doesn't have parents. This is found similarly to leaves but instead of checking for outward edges, check for inward edges with the command: graph.in_degrees(node) if it has none then it's a root.

```
agda_tree definition roots
```

Use Count

The use count query gets the number of times a definition is used. In other words, how many times does a definition appear as a dependency in other theorems. To find how many times a definition was used directly or indirectly is the same as counting the output of the dependents query. This query either accepts a -top=n option where it will return the top n most used modules or the -d=definition option that finds how many times a specific definition was used.

```
agda_tree definition uses -top=10
agda_tree definition uses -i -top=10
agda_tree definition uses -d
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
agda_tree definition uses -d -i
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Module Path To Leaf

This query is exclusive to the definition graph. The module path to leaf query gets the modules needed to get from one definition to another. This is done using the path to leaf query, but only keeping the modules of the path of definitions, repeats are removed.

```
agda_tree definition module_path_to_leaf
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Definition Type

This query is exclusive to the definition graph. The definition type query gets the type of the definition. This data is collected during the building of the definition graph then stored for each node.

```
agda_tree definition type "InfinitePigeon.Addition.n-plus-zero-equals-n"
```

Cycles

This query is exclusive to the definition graph. The cycles query gets the cycles in the graph, NetworkX provides a method to find simple cycles. Simple cycles are cycles where nodes aren't repeated, except for the start and end node. The method is: nx.simple_cycles(graph).

```
agda_tree definition cycles
```

Save Tree

This query is exclusive to the definition graph. The save tree query converts the graph into the DOT format. NetworkX allows for this conversion using the pydot library by using the method:

CHAPTER 7

Testing

Testing ensures that the tools behave correctly, it also allows for better maintainability as any changes can be automatically checked for issues. For Agda Tree, unit tests have been created for each query. This gives some guarantee that the queries are working as expected and that any future changes still return the same results. These tests were also timed, to measure the performance of the queries, while some will vary in performance depending on their input it gives a good estimate of their expected performance.

For Agda Comp, the performance of the compilation is the overall goal, so the testing is centred around the strategies. Checking that the strategies compile the projects correctly, such that no module is left not compiled and that a module isn't compiled at the same time as one of its dependencies. The time to create the compilation order with the strategy is also measured, but it is insignificant compared to the time to compile an Agda project.

7.0.1 Agda Tree Unit Tests

Unit tests have been created for each query and the options that the queries can take. These test use the TypeTopology definition and module dependency graph to test the results of the queries. While testing every combination of input for this graph and checking the output isn't possible, the test instead test one example of the use of that query. For example, to test the dependencies queries the definition "InfinitePigeon.Addition.n-plus-zero-equals-n" is used, and checked if the output is correct given a hard coded expected response. This is done for each query, along with their different options so for the dependencies query there is also a unit test for the indirect option. Integration testing while possible, isn't necessary as the CLI parsers are generated from the query functions. So the CLI and the queries don't need to be tested for their integration as the CLI is based of the queries directly.

7.0.2 Agda Tree Performance

The performance of the queries has been tested, although, this will be highly dependent of the size of the graph and the Agda project. In this case TypeTopology is a large project with about 50,000 definitions, this gives a good representation of how long the queries will take with a most projects. Most queries can be completed is less than a millisecond and the rest being completed in under 20 milliseconds. The exceptions being the cycles queries, finding how many indirect uses a definition has and finding the path between two definitions or a leaf. The cycles queries takes about 622 ms to be completed and the indirect uses takes about 3804 ms to be completed. Since the queries have recursively calls that traverse the whole graph, this is expected. The path between two definitions query can vary depending on the definitions, due to cycles and size of the graph the time to complete this query can go from less than a millisecond to impossible.

Lastly, the create tree query takes the longest time to be completed. As it has to compile the entire Agda project, then parse all the s-expression files into a graph. For TypeTopology this can take 10 minutes or more, although, this is mitigated as the command would only need to be run once and the graph can then be re-used.

7.0.3 Agda Comp Strategy Validation

The testing for Agda Comp centered around the correctness and safety of the strategies. For a compilation strategy to be valid it must be correct, that is it compiles all the modules in the project. It must also be safe, so a module can't be compiled at the same time as one of its dependencies or itself. These two properties are checked for the level strategy and the disjoint level strategy described in Sub Section **??**. The correctness check is done by adding each module and its dependencies to a set, if this set is the same as the set of all the modules in the dependency graph then it is correct as all modules are compiled eventually. The safety check is done by checking each step of the compilation order, if there are some index files that are compiled together then the dependencies of those index files are checked to be disjoint from each other. If they are disjoint, then they are removed from the graph and moves on to the next step in the compilation order. By the end if all index files compiled in parallel had disjoint dependencies then the overall compilation is safe.

The performance of Agda Comp is highly dependent on the Agda project, and will be assessed on the evaluation Section **??** as the purpose of the tool is to speed up compilation. The time to create the level and level B tests is 1-millisecond and level disjoint takes about 50 milliseconds. This time is insignificant compared to the 5 minutes it normally takes TypeTopology to be compiled.

7.0.4 Conclusion

Agda Tree unit tests each query, checking if a certain input gives the correct output. Making it easier to make changes while maintaining stability as the unit tests check that the output remains the same. The unit testes are timed, with most queries being negligible except for cycles, path between, path to leaf

and indirect uses which can take from half a second to unending depending on the input.

Agda Comp was tested on the strategies it used, whether the strategies were correct and safe. The Agda type checker doesn't give any perceivable error with unsafe or incomplete compilation, so these tests are critical to give a guarantee of a well done compilation. The time to create the compilation order is negligible compare to compilation time taking at most 50 milliseconds.

CHAPTER 8

Project Management

The timeline of the project can be seen in the below Gantt chart **??**. The focus on the first 2 months was on writing Agda Tree and Agda Comp. For Agda Tree, the first milestone is to parse the Agda project to extract the definition dependency graph. The next milestone is to implement the queries that analyse the dependency graph. Once the queries were implemented, the last milestone is the Command Line Interface created that gives the user a way to run those queries with their own parameters. Lastly, the queries were unit tested for correctness and tested for performance. This is done by the date of the demonstration, which is a major milestone as it marked the first stable version of the CLI tools.

While developing Agda Tree, Agda Comp is also being developed. The first task of Agda Comp is to explore the compilation strategies that could be used and different methods to compile an Agda project. Once the strategies are selected, they are tested for correctness, safety and compilation performance which is a major milestone. With the strategies implemented then, the CLI is created that automatically runs the strategy and the compilation. This is done by the date of the demonstration.

An agile software development methodology was used, as seen by the timeline there are multiple tasks running at the same time until they are all completed by the demonstration. There were weekly meetings with my supervisor were a working state of the project was shown along with a summary of all the accomplishments since the previous week, and my supervisor gives suggestions on changes and improvement for next week. This iterative approach allowed for the tools to be quickly created for my supervisor to test.

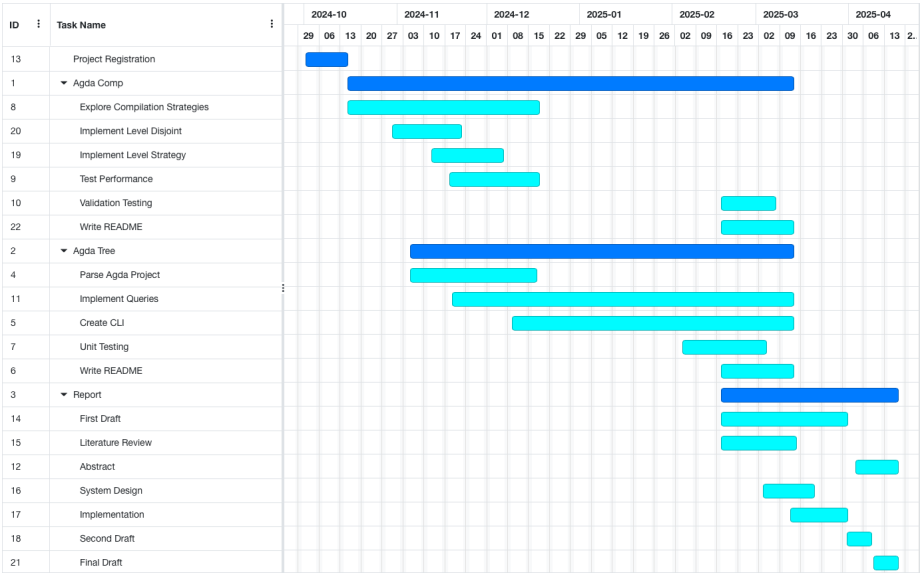


Figure 8.1: Gantt Chart

CHAPTER 9

Evaluation

This section evaluates the Agda Tree and Agda Comp tools with respect to the function and non-functional requirements set on Chapter 4. These requirements outline the purpose of these tools and their expected behaviour.

9.0.1 Agda Tree

Agda Tree extracts the definitions and their relationships from any Agda project, uses Agda to generate the module dependency graph and imports them into a NetworkX graph. It implements several methods that analyse this graph for information, meeting all the queries outlined for the definition graph in Table 4.2 and the module graph in Table 4.3. These methods are then exposed through a command line interface that lets the user runs those queries on their Agda project. The python library argparse validates that the input to the queries is valid. Then CLI the outputs the modules to standard output, which can then be piped to other utilities. The CLI will work with any Agda project, only the index files containing all modules needs to be passed in.

Agda Tree is able to extract s-expressions from any Agda project but depending on the size of the project and the user's computer it can take more than 10 minutes as seen on Section 7.0.2. Most queries are complete in under 2 seconds regardless of project size, but queries that require finding a path in the graph can take far longer depending on the size of the graph. The CLI is documented in the README and the `–help` options in the command, making the commands easier to learn and understand. Since Agda is mainly developed in macOS and Linux, the tools were tested in WSL and a MacBook Air without issues. If the CLI is provided with bad inputs it returns an error from NetworkX explaining what the issue was. The code base is documented through READMEs and unit tested allowing for better maintainability and testability. Therefore, Agda Tree meets all its functional requirement but doesn't fully meet its non-functional requirements due to performance issues. The requirements as described on Table 4.1 for functional requirements and Table 4.4 for non-functional requirements.

Limitations

A major limitation of the tool is finding the maximum path between a node and a leaf, depending on the node the query can run indefinitely. This greatly limits the usefulness of the query. Another limitation is the verbosity of the definition names and amount of definitions. The TypeTopology definition graph has more than 50,000 definitions but with further inspection not all these definitions provide valuable information for the user. Sometimes the definitions come from Agda's backend which the user has no use in seeing. The definition names become difficult to understand due to their length, and some definitions are ambiguous, so they require an ID number. Overall making find the name of a definition a bit cumbersome.

9.0.2 Agda Comp

Agda Comp effectively uses Agda to generate the module dependency graph and imports the module dependency graph into Python. Using this graph, Agda Comp allows for the user to pick between multiple strategies to compiles their project. The user can choose how many cores the compilation will use to limit the resource usage. The tool creates the necessary index files and make file, uses them to compile the project and then cleans up these files. The CLI has a README and a usage section with the –help options that make the commands more user-friendly and easy to understand. The argparse library validates that the correct options are passed in to the CLI or gives a user-friendly error message if not. The tool is able to discern the project structure from the index file the user passes in, making it usable across any Agda project. Therefore, Agda Comp meets most of its functional requirements except for speeding up compilation for any Agda project which will be explored in Section 9.0.2. The functional requirements were described in Table 4.5.

Agda Comp works for any sized project, it has a simple command line interface to choose the compilation strategy. It was tested within WSL and macOS, for compatibility. The codebase is documented with a README and the CLI has a –help option to help users better understand the tool. The strategies were tested to ensure correctness and safety. Therefore, Agda Comp meets all of its non-functional requirements as described on Table 4.6.

Limitations

This approach encounters two significant limitations, the first is that to create the module dependency graph it has to compile the project. Which means the only way this method could save time is if the user makes a change the doesn't change the module dependency graph allowing for a previous dependency graph to be re-used.

The second limitation is the overhead caused by the loading of the interface files. Interface files are the compiled Agda source code, when a module is type checked the information is stored in an interface file and used when compiling modules that depend on it. Every time a new process is created, it has to load all the interface files it requires to type-check a module and discard them once done, then the next process might load the same interface files again wasting resources. Meanwhile, if everything was compiled in one process like

normal it would only load the interface files once. This suggests that while parallelization could improve compilation time, it can't be done at the user level calling the Agda Type Checker multiple times. The Agda Type Checker must be parallelized from within, this would mean all the necessary interface files can be loaded once and the type checker can check modules in parallel with less overhead.

Compilation Strategies

The following Table 9.1 and Table 9.2 show the results of the testing of the compilation strategies. The compilation testing consisted of 7 tests of different strategies across 3 libraries. The three libraries picked are TypeTopology [10], unimath [18] and Agda's stdlib [19]. These libraries vary in size and methodology, for example unimath structure is of many small independent modules while TypeTopology has less but longer modules.

The 7 strategies tested are normal, which is the standard compilation which gives a baseline for the other modules. The unsafe test, which attempts to compile all modules in 4 index files at the same time, without regards to safety which shows the potential of parallelization. Then the level strategy using method A to split modules into index files described in Section **??**, the modules in the same level will be tested when split into 2 index files or 5 index files. The next test is using method B instead, where each index file has at most 2 modules or 5 modules. Lastly, the disjoint strategy is tested which has no parameters.

Tested on Agda 2.0.17, WSL has 16GB of ram with 8 performance cores at 5.4 GHz. Macos tested on Agda 2.0.17 has nGB of ram with n cores and n GHz.

Table 9.1: Results from WSL Testing Compilation Strategies

seconds (%)	Normal	Unsafe	Level A 2 cores	Level A 5 cores	Level B 2 cap	Level B 4 cap	Disjoint
TypeTopology	575 (100%)	280 (49%)	354 (62%)	355 (62%)	482 (84%)	394 (69%)	528 (92%)
stdlib	289 (100%)	147 (51%)	265 (92%)	243 (84%)	309 (107%)	261 (90%)	289 (100%)
Unimath	459 (100%)	219 (48%)	862 (188%)	362 (79%)	717 (156%)	644 (140%)	462 (101%)

Table 9.2: Results from Martin Escardo Testing Compilation Strategies macOS

seconds (%)	Normal	Unsafe	Level A 2 cores	Level A 5 cores	Level B 2 cap	Level B 4 cap	Disjoint
TypeTopology	345 (100%)	172 (50%)	287 (83%)	265 (77%)	344 (100%)	295 (86%)	369 (107%)
stdlib	189 (100%)	123 (65%)	241 (128%)	197 (104%)	231 (122%)	203 (107%)	203 (107%)
Unimath	302 (100%)	168 (56%)	863 (286%)	575 (190%)	633 (210%)	568 (188%)	304 (101%)

The results show that a time saving from compilation is dependent on the project it is compiling. TypeTopology benefitted the most from the compilations strategies achieving about a 40% faster compilation in WSL and 23% faster compilation in macOS. Although the results from the remaining Agda projects aren't as promising, with the macOS results showing no improvement in compilation time in any safe strategy. The results also show that the compilation time also depends on the user's system, with WSL achieving a speed-up in level A 5 cores with all projects.

The cause in the difference in compilation improvement per project, is likely due to the limitation discussed in the Section 9.0.2. Since unimath has more small modules, there is a large amount of interface files that are constantly being loaded and unloaded when normal compilation can load all the interface files at once and keep them.

9.1 Conclusion

Both Agda Tree and Agda Comp meets most of the functional and non-functional requirements set out in Chapter 4. Agda Tree effectively extracts definition dependency graphs from any Agda project, but it can take a long time to generate that graph. All queries are implemented, however some queries can take an indefinite amount of time to complete depending on the user input. One of the limitations of Agda Tree is the difficult to understand definitions and definition names, as the Agda backend doesn't make them user-friendly.

Agda Comp is able to import the module dependency graph from any Agda project and compile it using different strategies. Although, the effectiveness of those strategies to lower compilation time varies depending on the system and the Agda project. The limitation that to create the dependency graph the Agda project is compiled, means this tool could only be used safely when the user makes a change that doesn't alter the module dependency graph.

CHAPTER 10

Conclusion

Overall, Agda Tree is a CLI tool that extracts definition dependency graphs by using Andrej Bauer's s-expression extractor [3]. Parsing the s-expressions and importing them into a NetworkX graph. It also imports the module dependency graph from Agda. The CLI exposes queries on these graphs, that user that utilizes to better understand the code base. The CLI is user-friendly and easy to install for developers. Although the queries and accessibility are somewhat limited, due to the size of the Agda project and the verbosity of the Agda backend. The size of the Agda project can cause queries to take a significant amount of time and the Agda backend adds extra information that clutters the CLI output. Agda Tree achieved the project aim to be a tool that helps user understand large Agda codebases.

Agda Comp explores two strategies to compile modules in parallel, while maintaining safety and correctness. The first one is level sort, which sorts the modules into levels that can be compiled in parallel. The second being disjoint which finds large disjoint modules that are compiled in parallel. These two strategies were implemented into a CLI that allows the user to compile their projects by passing in the module they want to compile. Testing was done with these strategies using different parameters, across different Agda libraries. The tests showed that TypeTopology benefited the most from the speed-up while the other libraries got modest to no improvement.

This approach is limited in two ways, one being that having to call Agda's type checker multiple times means that the same interface files are being loaded multiple times. The second limitation is that to create the module dependency graph, the project must be compiled first which limits the use of the tool. Agda Comp doesn't fully meet its aim to speed up compilation time, depending on the system and project there could be no improvement.

10.1 Future work

Currently, it is cumbersome to search for definitions as the names aren't intuitive and become quite long. A better approach to deal with the definition names or a GUI could make the tool more intuitive to use. The definition dependency graph also contains a large amount of definitions that the user isn't interested in, finding an approach to reliably remove these definitions would improve user experience.

While Agda Comp shows the potential of parallelizing type checking and the possible time savings that could be made. In its current state it can't be used as a drop-in replacement and could be made more efficient. To address its limitations, the Agda type checker itself would need to be parallelized from within. This would avoid the overhead of loading interface files for every index file compiled, all the interface files could be loaded at once while the type checker works in parallel. To create the dependency graph the project must be compiled which severely limits the usage of the tool, an option to create the dependency graph without type checking would be incredibly useful. Ideally this would be implemented as a part of Agda, but the source code includes import statements for every module so exploring generating a module dependency graph from these import statements could help.

Bibliography

[1] agda. Github - agda/agda-language-server, Dec 2024. Language Server for Agda.

[2] Agda Developers. Agda documentation. `https://agda.readthedocs.io/en/latest/overview.html`.

[3] Andrej Bauer. Agda S-expression Extractor. `https://github.com/andrejbauer/agda/tree/release-2.7.0.1-sexp?tab=readme-ov-file`,.

[4] A. Bandi, B. J. Williams, and E. B. Allen. Empirical evidence of code decay: A systematic mapping study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 341–350, 2013.

[5] A. Bauer, M. Petković, and L. Todorovski. Mlfmf: Data sets for machine learning for mathematical formalization, 2023.

[6] K. Borowski, B. Balis, and T. Orzechowski. Graph buddy — an interactive code dependency browsing and visualization tool. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 152–156, 2022.

[7] ComputerHope.com. s-expression. `https://www.computerhope.com/jargon/s/s-expression.htm`.

[8] I.-A. CSÁSZÁR and R. R. SLAVESCU. Interactive call graph generation for software projects. In *2020 IEEE 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 51–58, 2020.

[9] M. Escardo. Mathstodon thread, Oct 2025.

[10] M. H. Escardó and contributors. TypeTopology. Agda development.

[11] R. Fairley. Tutorial: Static analysis and dynamic testing of computer software. *Computer*, 11(4):14–23, 1978.

[12] L. T. Gia Lac, N. Cao Cuong, N. D. Hoang Son, V.-H. Pham, and P. T. Duy. An empirical study on the impact of graph representations for code vulnerability detection using graph learning. In *2024 International Conference on Multimedia Analysis and Pattern Recognition (MAPR)*, pages 1–6, Aug 2024.

[13] N. Gunasinghe and N. Marcus. *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. Apress, 2021.

[14] Julian Vidal. AGDA$_H TML, Sept$.2025.

[15] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, Dec. 1999.

[16] R. R. Newton, O. S. Ağacan, P. Fogg, and S. Tobin-Hochstadt. Parallel type-checking with haskell using saturating lvars and stream generators. *SIGPLAN Not.*, 51(8), Feb. 2016.

[17] pappasam. Github - pappasam/jedi-language-server, Dec 2024. A Python language server exclusively for Jedi.

[18] E. Rijke, E. Stenholm, J. Prieto-Cubides, F. Bakke, and others. The agda-unimath library.

[19] The Agda Community. Agda Standard Library, Sept. 2024.

[20] universal ctags. Github - universal-ctags/ctags, Dec 2023.

[21] Y. Xiao, D. Park, A. Butt, H. Giesen, Z. Han, R. Ding, N. Magnezi, R. Rubin, and A. DeHon. Reducing fpga compile time with separate compilation for fpga building blocks. In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pages 153–161, 2019.

[22] A. Zwaan, H. van Antwerpen, and E. Visser. Incremental type-checking for free: using scope graphs to derive incremental type-checkers. *Proc. ACM Program. Lang.*, 6(OOPSLA2), Oct. 2022.

CHAPTER 11

Appendix

Some information, for example program listings, is useful to include within the report for completeness, but would distract the reader from the flow of the discussion if it were included within the body of the document. Short extracts from major programs may be included to illustrate points but full program listings should only ever be placed within an appendix. Remember that the point of appendices is to make your report more readable. I don't expect to see apprentices in any Project Proposals.