Todo list

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

Word count 12,500

Should I format Python library names differently?

Acknowledgements

I would like to express my gratitude to Andrej Bauer, the creator of the Agda s-expression extractor, and Job Petrovčič for their invaluable assistance in helping me use and understand the extractor. Their guidance contributed significantly to the overall success of this thesis. Their expertise and support have been greatly appreciated.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Add citations to articles about code decay and refactoring?

1.1 Background

When first encountering a massive code base with years of development it can be overwhelming and difficult to parse through [6]. On most libraries you can ignore its low-level functions, but in Agda it is useful to know the underlying definitions for future proofs. This creates the need for tools to improve this experience, such as using dependency graphs of code to find parts to refactor [9].

Agda is a functional programming language, that works as a proof assistant that follows the propositions-as-types logic system [2]. Since proofs in Agda are made 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 high-level definitions when making a proof, knowing how the types are fundamentally defined is beneficial.

1.2 Motivation

Due to the size of these projects and the complexity of the types it becomes difficult to fully grasp what the relationships between the definitions. This creates the need for a tool that can analyse Agda projects and give the user an easy-to-use interface to query these relationships and better understand the proofs. It would also benefit during refactoring which deals with code decay [14].

This tool would make it easier to refactor modules and change the structure of the project. For example, if a definition isn't used often, it can be removed, if a module depends on many other definitions it could be split into smaller parts. This information would give the user a better idea on where to put their focus.

In addition, large Agda projects have 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. Normally not an issue on high-level files, as Agda keeps track of the modified modules and only compile the modules changed and its dependents.

However, low-level files with many dependents, will cause a significant portion of the project to re-compile. This effect is particularly pronounced during refactoring where significant changes will be made to the structure of the project, causing Agda to re-check the whole project wasting time.

Agda's type-checking process checks module dependencies in sequential order. This presents the opportunity for parallel execution, given that modules can be compiled concurrently without dependency conflicts. There is potential for a speed-up. Therefore, a tool that finds modules which are safe to compile in parallel could lead to 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 by Andrej Bauer [3] that allows for a better view of Agda's internal representation. Although, S-expressions are not an ideal data structure for querying these definitions. They are structured for syntax and contain other information that can be discarded. A better data structure would be a graph, that contains the definitions as nodes are their relationships as edges. A tool that could explore this dependency graph of definitions, would let the user get a better graph of large code bases faster.

Also, Agda does have a command to create a dependency graph of the modules. For compilation time improvement, only 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. do not compile modules and their dependencies at the same time). Different strategies have to be tested against well-known libraries to gauge its effectiveness.

Explain further what the challenges are

1.4 Objectives

The objective of this project is to create a Command Line Interface (CLI) that generate the dependency graph from any Agda project using the s-expression representation [3]. Then the user access to queries that will let the user explore the graph and gain insight into the Agda project. There aren't any tools currently that have these features.

The second objective is to build a CLI tool that reduces compilation time. By automatically generating the module dependency graph from an Agda project. Applying a strategy to find the optimal order that modules should be compiled in parallel while maintaining safety. And lastly, running that strategy, compiling the entire project.

1.5 Project Structure

First, there will be an exploration of related works pertaining to using graphs for code analysis. As well as, an explanation of s-expressions, other instances of parallel computation and describing the scheduling problem. . Second, the requirements and overall design of the CLI tools will be own, along with an explanation of what strategic are going to be employed for compilation. Third, an explanation of the implementation of the designed systems. Fourth, an evaluation of the tools and the performance of the compilation strategies. Lastly, an overview of the results, limitations and future improvements.

do i need to explain the scheduling problem, how thoroughly?

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), and Code Property Graph (CPG). These representations encode the behaviour and properties of a code base, they can have many uses such as for code vulnerability detection [15]. These representations also help with static analysis, where code isn't run, but the structure is analysed for software validation [13]. For this project these graphs encode too much information, what is needed is a dependency graph only with definitions are their relationships.

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 which lets the user jump to where a definition is defined or referenced. The LSP sits between the code editor and the language server, the language server analyses the code to support these features [16]. Language servers have 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 [20] and Agda's language server [1].

Ctags [23] is a tool that indexes all the symbols in a project, this is helpful to get all the definitions from an Agda project. It helps the user with code comprehension and gives the text editor more information to improve the developer experience. However, it doesn't capture the relationship between the symbols which is a key aspect of a dependency graph.

Graph Buddy is an interactive code dependency browsing and visualization tool [7]. It takes large Java codebases and turns them into Semantic Code Graphs (SCG), and creates a visualization of this graph. It 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. This is quite similar to the goal of this project, it contains all the definitions and their relationships. Furthermore, the visualization also makes simpler to use than a CLI. However, it is exclusive to Java.

There is also a tool [10] 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. Helping with developer experience and understanding how the code flows through the project. This isn't relevant to Agda, as a proof assistant it helps with proving theorems which doesn't fit with call graphs.

2.2 Parallel Compilation

Is there any related work that I could add?

Type-checking is a computationally expensive sequential task that hinders workflow during development, so parallelizing could lead to a speed-up like in other applications. 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. Both can reduce compilation time, improving the developer experience.

An example is the work by Newton et al. [19] which seeks to parallelize type-checking with Haskell. Also, the work by Zwaan et al. [25] using scope graphs for incremental type-checking. These tools give methods to improve type checkers for general applications, there isn't a focus on Agda.

This project doesn't aim to optimize the type-checking algorithm itself, 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 [24].

Add section explaining scheduling problem?

Fundamentally the compilation problem in this project is a scheduling problem, which is NP-complete[18]. This means that there many algorithms that attempt to tackle this problem by following different assumptions as shown by Yves Robert [18].

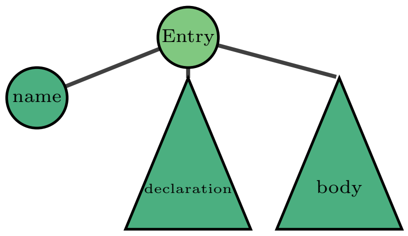
2.3 MLFMF: Data Sets for Machine Learning for Mathematical Formalization

MLFMF is a collection of data sets for benchmarking systems that help mathematicians find relevant theorems when proving a new one [5]. The data sets are created from large libraries of formalized mathematics in Agda and Lean. They represent each library as a graph and as a list of s-expressions. Some of the libraries included are as Agda-unimath and TypeTopology. The collection is a base for investigating machine learning methods to mathematics. The methodology to extract s-expressions can be used with other libraries to continue expanding the 250000 entries in the data sets.

The s-expressions extractor is created in Agda by extending the backend [3]. The backend has access to internal information about a project's definitions and its connection with other theorems. Andrej Bauer used that backend and converted the internal information into easy-to-parse s-expressions. The Figure fig. 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 its implementation. Mind that `(...)` are more s-expressions that were replaced for readability. A more general structure to the tag can be seen in fig. 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 important, as Agda by itself doesn't provide a convenient method to find all the definitions of an Agda project along with its implementation. This extractor packages the important information into a format that is easy to parse. Andrej Bauer made the s-expression extractor open-source in a GitHub repository [3].

This paper also describes another graph that can be generated from these s-expressions, this graph contains all the information about each theorem and what definitions it uses and how. While this graph could be queried by the user, it contains a too much information that is unnecessary for the user to explore.

2.4 Conclusion

There are many tools that analyse and visualize the structure of programming projects. They are used in static code analysis, where software can be validated and developers can use them to explore a project. However, most of the tools are not easy to query by the user, are meant to be used while editing a file or are reserved for more popular languages like Java. A tool that can read an Agda project and give access to the underlying graph is 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. Which is then converted into a graph that contains all the details of the Agda source code.

Slow compilers are a common problem which hurts developer experience. Due to the monolithic nature of type-checker, parallelization becomes a route to reduce 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 licences 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 analyse 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

Should I include personas?

The requirements outlined in this section, provide a target for the CLI tools to meet, in order to be useful to its users. This outline ensures that the CLI tools are easy to install by any developer, they need minimal extra knowledge to use the tool. The interface must be intuitive and there should be little friction between the development workflow and its use. 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 has to work with any Agda project and the user should be able to install the tool and its dependencies easily.

4.1.1 Functional Requirements

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

Table 4.1: Agda Tree Functional Requirements

ID	Name	Description
1	Definition Dependency Graph Extraction	Parses Agda projects and constructs a definition dependency graph
2	Querying the Definition Graph	Allows the users to query the dependency graph and retrieve information. (See table 4.2 for queries)
3	Command-Line Interface	User-friendly CLI that queries the dependency graph
4	Input Validation	Validates user input and provides clear error messages for invalid inputs
5	Integration with Agda Projects	Agda projects are structured differently, all valid structures are supported
6	Output Generation	Displays the query results in a readable format that follows the style of other Unix CLI tools
7	Module Dependency Graph Extraction	Parses Agda projects and constructs a module dependency graph
8	Querying the Module Graph	Allows the users to query the dependency graph to retrieve information. (See table 4.3 for queries)

Martin Escardo asked in Mathstodon [11] for possible queries that this tool should implement. The tool queries 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 graph are mostly the same as above except the module graph is a directed acyclic graph (DAG) giving it some 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 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 respond quickly. Agda is used mainly in 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	Extracts the dependency graph in 10 minutes depending on the size of the project
2	Query Performance	Responds to a query in under 2 seconds
3	Scalability	Allows for fast querying of large projects
4	Usability	Easy to use, with intuitive commands, clear documentation, and meaningful error messages.
5	Compatibility	Works on macOS and Linux
6	Reliability	Handles bad inputs gracefully
7	Maintainability	Well documented and well-structured to allow for new queries
8	Testability	Tested to ensure queries give the correct output

4.2 Agda Comp

The Agda Comp CLI tool, is an application that runs on the terminal. It extracts the module dependency graph from an Agda project, produces the order in which the modules should be type-checked and proceeds to compile the project with that order.

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 works automatically, the user only needs to input what they want to compile. It 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 a module and its dependencies concurrently.

Table 4.5: Agda Comp Functional Requirements

ID	Name	Description
1	Module Dependency Graph Parser	Parses Agda's dot file module dependency graph
2	Compilation Strategies	Selection of compilation strategies to apply
3	Compilation Customization	Customization of parameters for the compilation strategy (i.e. amount of cores used)
4	Compilation	Creates index files and a make file that will safely compile all modules
5	Command-Line Interface	User-friendly CLI with commands to run and customize compilation
6	Input Validation	Validates user input and provides clear error messages for invalid inputs
7	Integration with Agda Projects	Agda projects are structured differently and are all supported
8	Speed Up Compilation	Compiles Agda projects faster than a normal compilation.

4.3 Non-Functional Requirements

Agda Comp is a tool that slots in next to Agda seamlessly, so users who already have Agda do not need extra setup. The tool is easy to use and works with any Agda project. Agda is not developed for Windows, so the main focus is for the tool to work in a Linux and macOS environment.

Table 4.6: Agda Tree Non-Functional Requirements

ID	Name	Description
3	Scalability	Allows for compilation of large projects
4	Usability	Easy to use, with intuitive commands, clear documentation, and meaningful error messages.
5	Compatibility	Works in macOS and Linux
7	Maintainability	The codebase is well documented and well-structured to allow for new strategies
8	Testability	The compilation strategies are 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 and outputs useful information to the user. All the queries an Agda developer would need are included or simple to add. The non-functional requirements define how the Agda Tree must behave, for it to slot into a developer's workflow without issues.

The functional requirements for Agda Comp show the features needed for the tool, it must work with any Agda project and give a choice on how to compilation given some parameters. The non-functional requirements define the behaviour 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 a module and its tendencies at the same time. Since the time it takes to compile varies on the project size, performance measures aren't going to be made.

CHAPTER 5

Design

The design section describes the 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 is created manually.

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

5.1 Agda Tree

Agda Tree is a CLI that lets the user interact with the module and definition graphs. The first command designed is how the both dependency graphs are created. fig. 5.1 illustrates how the user will interact with the CLI to create the graphs. The user provides the Agda file that they want to analyse, normally this would be the entire everything index file. The everything index file is the file that imports all the modules in a project. This is standard in Agda, as this is the only way to type-check the whole project. Depending on the Agda project, this file will be generated automatically or by the project maintainers.

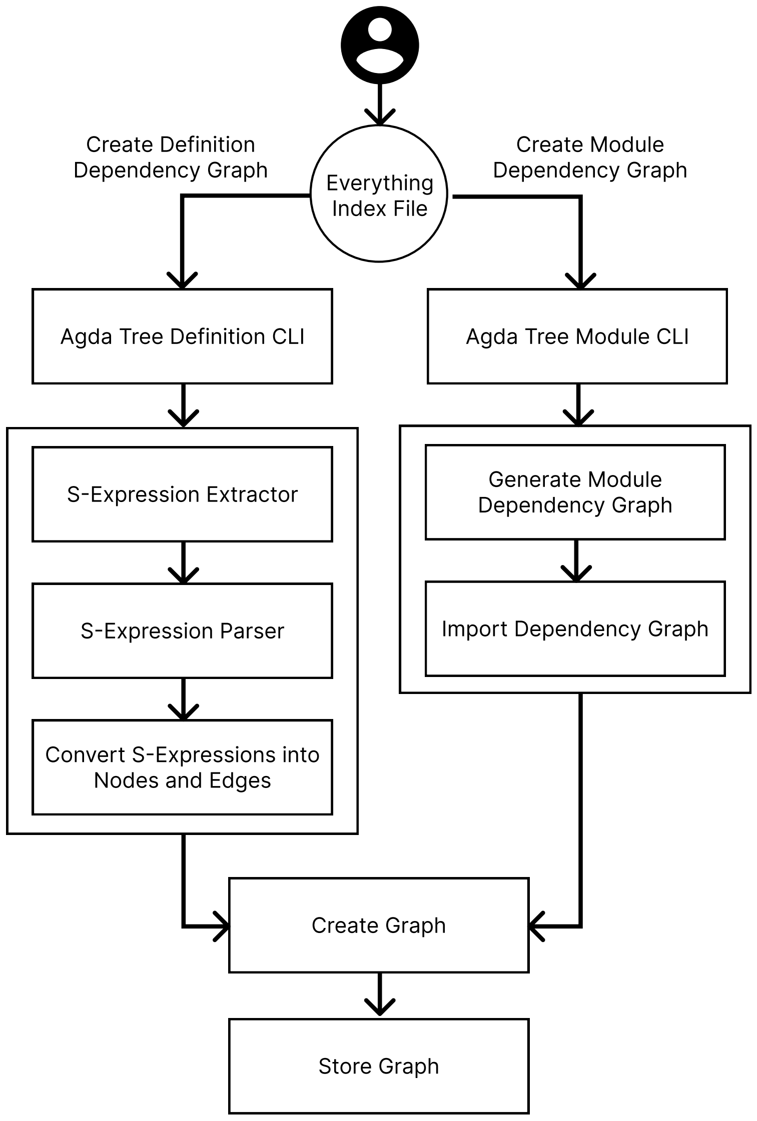


Figure 5.1: Agda Create Tree Diagram

fig. 5.2 is a Class Diagram, showing the classes and methods to operate the command line. The program starts with the command line's entry point, that will parse the input by the user and delegate the query to the respective dependency graph. Each dependency graph allows for a different set of queries. These queries can be found in table 4.2 for the definition graph and in table 4.3 for the module graph.

The definition graph creation depends on the s-expression extractor and parser. They read the Agda projects and convert the data found into the dependency graph.

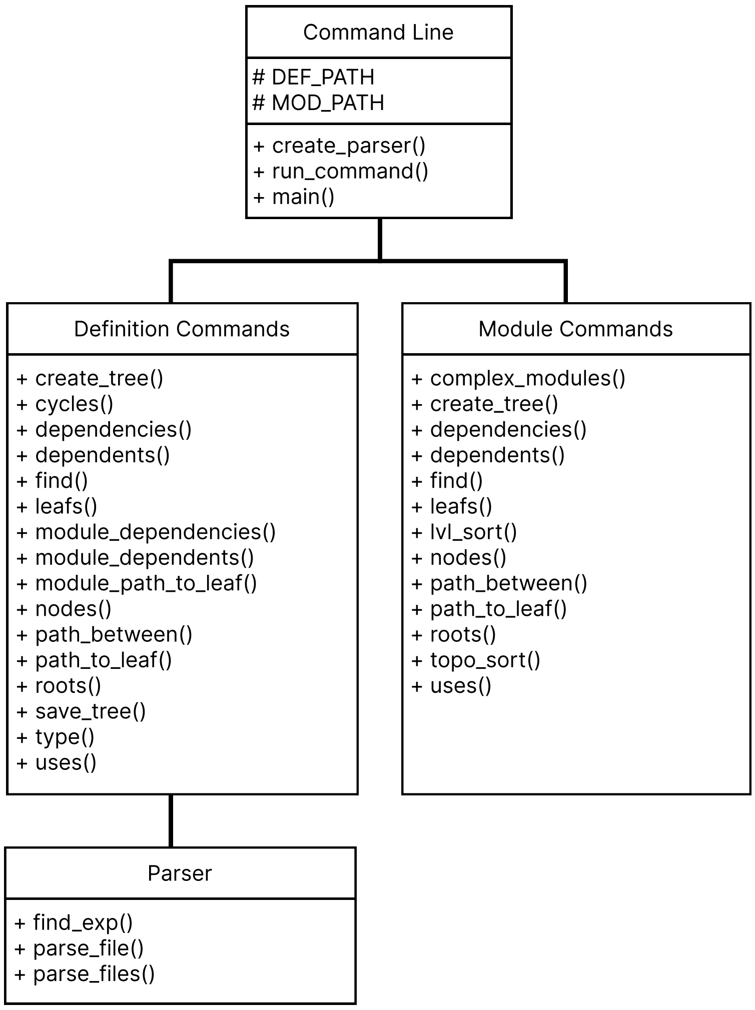


Figure 5.2: Agda Tree Class Diagram

fig. 5.3 shows how is split CLI into two, depending on what dependency graph is being queried. When the user makes a query, they will select the dependency graph and the respective queries will perform that query. The output will be displayed in stdout, which can be used with other operations like piping and xargs.

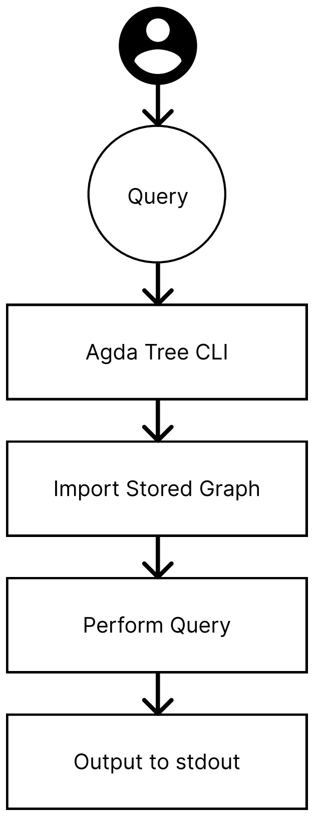


Figure 5.3: Agda Tree Query Diagram

5.2 Agda Comp

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

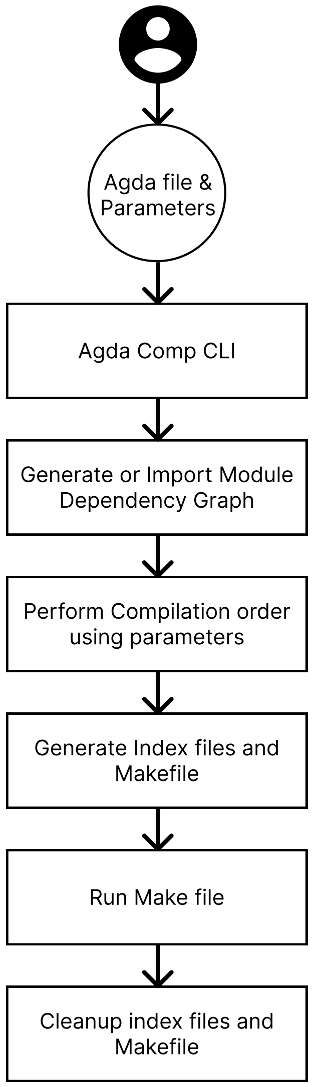
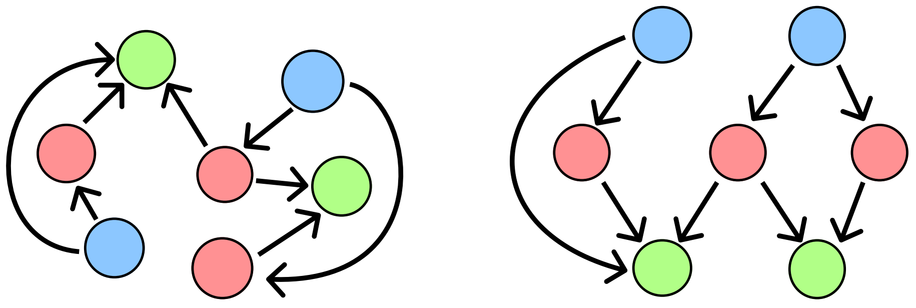

Agda file &
Parameters
Agda Comp CLI
Generate or Import Module Dependency Graph
Perform Compilation order using parameters
Generate Index files and Makefile
Run Make file
Cleanup index files and Makefile

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 fig. 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

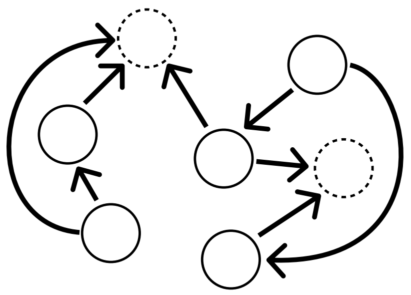
Figure 5.5: Level Strategy Example

This sorting has the property 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, 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 an 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 combined dependencies in previous levels, then compile the two modules in parallel. This becomes significant during implementation section 6.2.1, 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 targets 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 and tries finding large modules again. Note that once a module is compiled, it is no longer a dependency of future modules. This process is visualized below on fig. 5.6.


(a) Example dependency graph

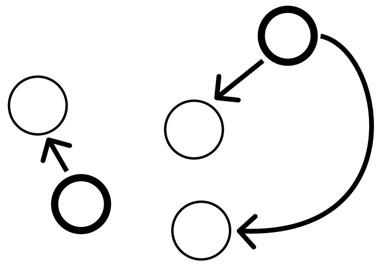

(b) Example dependency graph sorted where the leafs were removed, creating two modules that can be compiled in parallel.

Figure 5.6: Level Disjoint Example

This strategy ensures that at each step the amount of modules to compile is reduced. The strategy only compiles two modules in parallel if their dependencies are disjoint, if such modules aren't found then the leafs are compiled which do not 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 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 section 6.2.1. The disadvantage is that finding these disjoint modules is difficult, projects have hundreds of modules and finding multiple modules that do not share dependency can't be done through brute force. Each Agda project also has different structures with Agda-unimath [21] having smaller independent modules while Type-Topology [12] 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 fig. 5.2 show how the structure of the program. fig. 5.1 and fig. 5.3 also shows how the dependency graphs will be created and how the user will interact with CLI. This structure is important to ensure 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 fig. 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 lets the user to easily apply the compilation strategies which are modelled in section 5.2.1 and in section 5.2.2, then are implemented in chapter 6.

CHAPTER 6

Implementation

The Agda Tree creates a dependency definition graph from an Agda project and lets the user query this graph through a CLI. First, a subsystem is made that turns Agda projects into a dependency graph is implemented. Then, a CLI is created that lets the user query the graph. These subsystems must meet the functional requirements described in table 4.1 and the non-functional requirements described in table 4.4. The CLI implements 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

Agda Tree uses the s-expression extractor [3] to get the definitions and relationships from the project. These s-expressions are parsed and converted into a definition dependency graph. Which is then exposed through a CLI that the user can query. Agda already includes a command to generate the module dependency graph, so it only needs to be imported and attached to a CLI.

6.1.1 Building the definition tree

Agda's backend has the information needed to find the definitions and their dependencies, in an Agda project. Natively there isn't a way to retrieve this information 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 works in the backend, and converts Agda source code into s-expressions [3] which are easy to parse.

S-expression extractor

The s-expression notation is used in Lisp programming languages, it represents programs and data as tree-like data structures [8]. 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...)	Root tag that holds the whole module, module-name is the name of the modules and entries are the definitions in the implementation
(:module-name name)	Module name
(:entries name type body)	Definition, it includes its name, type and the body of the definition
(:name name)	Name of a definition, this name can appear as the name of an :entry tag, within the :type or :body tag
(:type type)	Description of the type of the parent definition
(:body body)	Body of a definition

Note that the s-expression extractor is a modified version of Agda, 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 6.1.4 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 complete detail, but this information is not needed. The information needed is the definition names and its dependencies which means the `:name` and `:entry` tags are of interest.

The s-expressions are compiled and saved into a directory. Then using Python and the library sexpdata is used to load the raw s-expression files into lists. These lists are traversed recursively to find the relevant tags. The methodology is to find all the `:entry` tags, 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 each key is the definition name and the value is a list of :name tags. Since an `:entry` tag represents a definition and the `:name` tags are the names of the definitions 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 traverses an s-expression, keeping track of the occurrences of a given tag.

The s-expression extractor writes the s-expression to a file per module, so the mentioned dictionary is created for each file and combined into one dictionary with all the definitions. The same process repeats, instead of looking for `:name` tags inside the `:name` tags in the `:type` tag instead of `:entry` tag. Store this in a separate 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 built in section 6.1.1 forms the definition dependency graph, with keys as nodes and values as edges. It is imported into the NetworkX library. This library efficiently creates and queries graphs, it implements useful features and is performant. NetworkX is a widely used tool, so a user can become familiar with its use and implement their own queries.

Do I need to explain the code this specifically

The graph is first initialized with the command `nx.Digraph()`, a directional graph. Then the definitions are added as nodes with the command `graph.add_nodes_from`, with the keys of the dictionary passed in. Lastly, the edges are added with the command `graph.add_edges_from` where an array is passed in containing tuples with the definition and its dependency. Note that when creating the edge the definition is the first, and it's the dependency 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', Pickle is 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 names include their module path along with an identifier number. Due to the way the s-expressions are extracted and the way Agda works internally two distinct definitions will have the same name with identifier to distinguish them. 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

Do i need to mention what commands were used to create the graph?

Agda has a built-in feature that creates a DOT file of the module dependency graph of an Agda project. This command is `agda --dependency-graph=[PATH] [Index File PATH]`. The DOT language describes how to describe nodes and edges for 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. Once Agda generates the DOT file it can be imported into NetworkX where it can be queried. This graph will then be pickled and stored for future use.

6.1.3 Commands

Most queries are the same between the definition graph and the module graph, but the module graph is acyclic giving it different properties. These queries can be explored in further detail in its repository [17].

Should I trim this, or keep all the queries

Show output of commands?

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
```

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 a pattern. The user provides a regex pattern, 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 defined in.

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

Dependencies

The dependencies query gets all the dependencies of a definition. This means the theorems the definition needs to be implemented. 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 their children 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 are theorems that use this definition. 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 their parents 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 node and counting its outward edges with the command `graph.out_degree(node)`, if the count is zero they are a leaf.

```
agda_tree definition leafs
agda_tree definition leafs -c
```

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 modules of the dependencies. To avoid repeated modules from they are added to a set. This command has an option for the indirect dependencies.

```
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 then add them to a set to remove repetition. This command has an option for indirect dependents.

```
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 leafs 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 largest one is returned.

The definition dependency graph cyclic while the module dependency graph is acyclic, this causes a difference in performance 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 counting for outward edges, count the inward edges with the method `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"
```

Type

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

```
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: `nx.nx_pydot.write_dot(graph, path)`.

```
agda_tree definition save_tree "/tmp/definition.dot"
```

Path Between

The path between query finds the longest path between to definitions. NetworkX provides the method: `nx.all_simple_paths(graph, src, dst)`, were given two nodes it will return all the simple paths between them. Simple paths are paths that do not repeat nodes. After finding all the paths, it measures their lengths and returns the maximum length.

```
agda_tree definition path_between
    "InfinitePigeon.Addition.n-plus-zero-equals-n"
    "MLTT.Natural-Numbers-Type.N"
```

Level Sort

This query is exclusive to the module graph. The level sort query sorts the modules into levels based on how far they are from a leaf module. This is done recursively, where the level of a node is based on the maximum level of its children plus one. This sorting is explained in section 5.2.1.

```
agda_tree module lvl_sort
```

Topological Sort

This query is exclusive to the module graph. The topological sort query sorts the modules into a topological order. Topological sort orders the modules into list where a module only depends on previous modules in the list.

```
agda_tree module topo_sort
```

6.1.4 Command Line Interface

The CLI uses Python, as it is a popular language that most users already have installed and may have some experience with. Making it easier for users to add new queries and make the changes.

Python includes a library called argparse to create CLIs. The user inputs are stored in `sys.argv`, argparse parses it based on the defined parsers and creates help with usage messages. There is the main parser which contains two sub parsers, one being for the definition queries and the other being for the module queries. The main parser decides whether the definition graph or module graph is used. Then it is delegated to the respective sub parsers.

Do i need to go into this much detail

How do i format file names

The sub parsers are generated automatically from the methods that implement the queries. There are two files `def_cmds.py` and `mod_cmds.py` which store the functions that perform the queries. The functions in this file are read, and are used to automatically generate the CLI. This allows for greater flexibility as to add or change a query, only the function has to be changed, and the interface will update by itself.

This is done with the included Python library inspect, which can get the functions in a file, get their parameters and their documentation. The way the subparsers are generated is by first getting all the functions names which are the queries. For each function, the parameters it requires will be the input from the user. If it is a position parameter then the user must give it, otherwise, it is an optional parameter. The inspect library can also read the documentation of a function, if a comment is made below a function it is read as documentation which is added to the help description of the query. The documentation for each parameter has to be done manually, where a dictionary with the parameter name and its description is used to give each option in the CLI a description. Agda Tree's help messages can be seen in listing A.2 and listing A.3.

For example the function:

```
def dependencies(g, d, indirect=False):
    """Definitions that definition d depends on, -indirect will find the
    indirect dependencies"""
```

becomes:

```
agda_tree definition dependencies -h

usage: agda_tree definition dependencies [-h] [-g G] [-indirect] d

positional arguments:
  d           Definition name

options:
  -h, --help  show this help message and exit
  -g G        Path to tree (Default: ~/.agda\_tree/def\_tree.pickle)
  -indirect   Get indirectly connected nodes
```

Once the parser is created, the user input is validated then the query executes. To execute the query the appropriate graph is loaded, assuming the graph to already exists. Since the name of the query is the name of the function and Python allows for a function to be called with its name as a string then it is used run the function. The parameters of the function are the same as the input from the user, so this is passed in directly. The output of the function is then printed to standard output.

It is important for the output to be printed to the console, then the output can be piped into other terminal applications and be used in conjunction with other commands.

6.1.5 Installation

For the tool to be distributed and installed by the users, it is packaged. The project is packaged using the `pyproject.toml` file which describes the metadata of the project. The Hatchling backend was chosen to create the distribution that builds the tool for other computers. The project file also contains the project dependencies along with the Python version, this file is used to build the project and locally install it.

PIP is Python's package manager, it can install packages from an online repository or locally so with the project file installing this tool is as simple as running `pip install .` . However, for end-user applications it is better to install using PIPX which isolates the environment of the tool from the remaining system, to avoid clashing package version. PIPX can be installed by the user with their respective package manager.

6.1.6 Installing Agda S-Expression Extractor

Should I explain s-expression extractor installation

The Agda S-expression extractor is an extension of the Agda backend by Andrej Bauer [3]. This extension isn't pre-built for a user to download, it must be built. To make the Agda Tree easier to install, it includes a script to install it.

This script clones the repository into a temporary directory and checks out the branch to the latest version of Agda (2.7.0.1). Using Stack, a tool to build Haskell projects and manager their dependencies, to build it. Stack can be installed through most package manager, although Agda is written in Haskell, so Agda Developers likely already have this tool installed. The repository contains a YAML file with instructions for Stack to build the binaries. The binary is then copied to the `/.local/bin/` folder, which the user already has in their environment path.

One the binary is in the path, it can be accessed through the command `agdasexp`, Agda Tree will recognize this command and use it to build the definition graph.

6.2 Agda Comp

Agda Comp automatically creates the module dependency graph as described in section 6.1.2, using Agda's built-in command. The graph is passed into a strategy, the strategy will return the order in which to compile the modules. Given this order, a 'make generator' sub-system creates the index files and the make file which compile the modules in parallel following the order.

6.2.1 Strategies

Each strategy will take the module dependency graph as a NetworkX graph where each node is a module and each edge is an arrow starting the at the module and pointing towards its dependency. To compile a module, its dependencies must be compiled first. Also, two modules can't be compiled at the same time as this would be inefficient and could cause issues with two files being written at the same time. A valid strategy should compile all the modules in a project and do it safely so without compiling a module and its dependency at the same time.

The strategies output a 3D array describing the order in which to compile the modules and which modules can be compiled in parallel. An index file is a collection of modules that are compiled together, not in parallel. If two indices have disjoint dependencies then those two indices can be compiled in parallel. The array returned by the strategies is a list of the order in which indices are compiled together. For example, `[[[module 1 , module 2] , [module 3, module 4]], [[module 5]] , [[module 6, module 7]]`. What this array shows is that `module 1` and `module 2` should be compiled together in an index, the same for `module 3`, `module 4` and `module 6`, `module 7`.

Then they themselves are within a list meaning that these to indices should be compiled in parallel. So another way to write the array is the following: `[[index_0_0 , index_0_1], [index_1_0] , [index_2_0]]`. Where `index_0_0`, `index_0_1` are compiled in parallel, the `index_1_0` is compiled by itself then `index_2_0` is compiled by itself in that order. When Agda compiles a module, it has to load in interface files that are the compiled version of the modules it depends on. By adding as many modules into one index, means that the interface files only have to be loaded in once limiting overhead. With this array, the make generator subsystem will create the index files and make file that will compile the indices in order.

Level Strategy

As explained in the design section 5.2.1, the level strategy works by sorting the modules into levels. Where the level of a module is its maximum distance from a leaf, guaranteeing that if previous levels are compiled then the modules in the current levels can be compiled in parallel.

The sorting algorithm, groups modules into levels where the level is the maximum distance of a module to a leaf, the output is stored in a dictionary. The level of a module is the same as the maximum level of its children plus one, so the algorithm recursively calls the function with it's the module's children. The dictionary stores previously analysed modules to avoid repeated work. This algorithm is shown in listing A.4

The compilation order array is created from this sorting, where each level is a list of modules that can be compiled in parallel. Then the make file is generated from this compilation order as explained in section 6.2.2.

A practical detail to note is that compiling modules individually causes a large amount of overhead, as for Agda to type check a module it has to load interface files of the module's dependencies which can be large. This loading process will occur every time a module is type-checked, to avoid this overhead the modules in the same level are grouped into index files. This way the loading of interface files only happens once per index file.

There are two ways to split the modules in to index files, method A is to split the modules evenly across n index files were n would be analogous to the cores of a system. A performance increase could only happen if the computer is doing the compilation work in parallel on separate cores, so creating more index files than cores would cause more than 1 index file being compiled in the same core.

Method B is to cap the amount of modules in an index file, if the cap is exceeded another index file is created. While this will create more index files than cores that can compile in parallel, due to the overhead of loading interface files, being able to type check one index file while waiting for the data of another could still result in a performance increase. These two methods will be tested on the evaluation section 9.2.2 as level A and level B.

Level Disjoint Strategy

As explained in section 5.2.2, the level disjoint strategy finds the largest disjoint modules, modules with a large amount of dependencies which are all distinct. If the algorithm finds the disjoint modules, it compiles them and doesn't consider them further. Otherwise, it compiles the leafs removing common dependencies. Eventually the algorithm will terminate as there aren't any modules left.

The challenge is in finding the disjoint modules efficiently, large projects can have hundreds of modules, so a brute force approach isn't feasible. Thus, a greedy approach was taken. For two modules to be disjoint from each other, that means that the leafs they indirectly/directly depend on must also be different. The greedy algorithm sorts the modules in ascending first by the amount leafs a module depends on, then breaks ties by putting the module with the largest amount of dependencies first.

Using that sorting the greedy algorithm stores the modules into buckets. Each bucket has modules that depend on a set of leafs. The first item of the sorted list creates a new bucket which is added to it. Then the algorithm loops through the remaining modules in the sorted list, if the module depends on distinct leafs from previous buckets a new bucket is created, and it is added to it. If a module depends on the same leafs as one of previous buckets, then it is added to it. But if a module shares only some but not all the leafs it depends on with another bucket, it is ignored.

The compilation order array is created by first adding the list of leafs to be compiled as their own step, then the buckets are added as a step that can compile each bucket in parallel. These bucket approximate the most amount modules that can be compiled in parallel without conflict. Eventually all modules would have been added as a leaf or a bucket to the compilation order and the algorithm terminates.

Sometimes this algorithm will not be able to compile in parallel if only one bucket is found, in these cases the leafs and the buckets are combined into one compilation step.

6.2.2 Building The Make File And Indices

An index file is a module that imports other modules but has no definitions within it, this is done to type check a collection of modules. The `.lagda` file extension is used for the index files, meaning latex like syntax is used to write the module. Example index file from TypeTopology:

```
Generated Index file

\begin{code}
   {-# OPTIONS --without-K --type-in-type --no-level-universe
        --no-termination-check --guardedness #-}
   import MLTT.Universes
   import MLTT.Natural-Numbers-Type
   import InfinitePigeon.Logic
   import Various.UnivalenceFromScratch
\end{code}
```

It starts with flags that changes the behaviour of the type checker, for example not allowing the imports of incomplete modules. These flags are going to be different for all projects, so the flags are scraped from index file passed in by the user. Then, code environment is opened, and each module is imported. The environment closes and nothing else needs to be added.

The index files are named based on the order in which they are compiled, so index files `index-0-0` and `index-0-1` are compiled first and in parallel, then `index-1-x` is compiled next and so on. This allows for the Make file to be generated intuitively.

Example:

```
all: _build/2.7.0.1/agda/source/index-1-2.agdai
```

```
_build/2.7.0.1/agda/source/index-0-0.agdai:
  agda ./source/index-0-0.lagda

_build/2.7.0.1/agda/source/index-1-0.agdai:
    _build/2.7.0.1/agda/source/index-0-0.agdai
  agda ./source/index-1-0.lagda

_build/2.7.0.1/agda/source/index-1-1.agdai:
    _build/2.7.0.1/agda/source/index-0-0.agdai
  agda ./source/index-1-1.lagda

_build/2.7.0.1/agda/source/index-2-0.agdai:
    _build/2.7.0.1/agda/source/index-1-0.agdai
    _build/2.7.0.1/agda/source/index-1-1.agdai
    _build/2.7.0.1/agda/source/index-1-2.agdai
    _build/2.7.0.1/agda/source/index-1-3.agdai
  agda ./source/index-2-0.lagda
```

A makefile describes order of commands that need to be run to compile a project. Makefiles are made of targets, how to build them and the prerequisites for building them. In the example, to compile `index-1-2.lagda` then `index-1-0.lagda` and `index-1-1.lagda` must be compiled first. The `all:` target is the entry point to compile the entire project meaning compiling the highest level index `index-2-0.lagda`. Agda compiles modules into interface `.agdai` files, the Makefile will check if interface files exist, if it doesn't, it attempts to build it otherwise it will continue to the next prerequisite. If the Makefile contains two targets that do not depend on each other, it will automatically attempt to compile them in parallel. In this case `index-1-0.lagda` is compiled in parallel with `index-1-1.lagda`.

The make file is made naturally from the output of the compilation strategy, where the targets are the index files (named after their position in the array) and the prerequisites are the index files that came just before it in the array. Once all the indexes are in the Makefile, then the last index that needs to be compiled, that depends on all the previous modules is added to the `all:` target. Note that the first index files have no dependencies, so there are no prerequisites.

6.2.3 Command Line Interface

The CLI is written in Python, it is a popular language that many people already have. Similarly to Agda Tree CLI section 6.1.4, argparse for the CLI. argparse parses the user input and generates usage/help messages. The parser accepts the path to the modules file to compile as a positional parameter, then other options can be passed in to change what strategy to use. The help message showing these options is show in listing A.1.

There are three options, clean, jobs and strategy. By default, the tool re-uses the previous module dependency graph and removes already compiled modules from the graph. The clean option generates the module dependency graph again and deletes the `_build` directory which contains the compiled modules. Note that Agda Comp finds if a module is already compiled and unmodified is by checking whether its corresponding interface file exists and wasn't modified after its creation. If a module was changed after then it is considered modified and will be re-compiled. The jobs option sets how many cores are used in the strategy and compilation. Lastly, the strategy option lets the user pick which strategy to use.

When the user provides the index file and sets the options they want, the index files and make files are generated. Using the path of the index file, the make files and indices are moved to the project root directory and the project source directory respectively. Then the command `mk compilation.mk` is run, where `compilation.mk` is the name of the generated make file. This will compile the project using the cores given by the user. Once finished the index files and make file are deleted and the time to compile is printed. Also, if the user chooses to cancel the compilation there is a listener that will delete the make files and index files as to not pollute the user's project.

For example the following command will compile TypeTopology using 2 cores, it will delete previously compiled modules, recreate the module dependency graph and use the level B compilation strategy.

```
agda_comp -j=2 -c --strategy=levelb
    "/tmp/TypeTopology/source/AllModulesIndex.lagda"
```

6.2.4 Installation

The installation process for this tool is the same as for Agda Tree in section 6.1.5. A pyproject.toml file that describes the dependencies, backend and python version of the project along with other metadata. This packages the project such that it can be installed through PIP, although PIPX is recommended to install end-user applications as PIPX will isolate the dependencies of Agda Comp from the packages of the user.

6.3 Conclusion

Agda Tree uses Andrej Bauer's s-expression extractor to create the s-expression files, which are then imported as lists into python using the sexpdata library. These lists are explored to find all the definitions and their relationships which are then converted into a graph in NetworkX. Then the CLI exposes queries that can be made to the dependency graph for the user to learn more about the relationship of these definitions. The tools are packaged to be easy to install and the building of the s-expression extractor is handled.

Agda Comp uses different strategies to analyse Agda's module dependency graph to improve compilation time. The strategies read the dependency graph and return a list that shows the order in which to compile the modules. With this order, index files and a make file is generated that will compile the modules in the correct order. The make file compiles the project, then deletes itself along with the index file to leave the user with a clean project. Agda Comp can use different strategies and change the amount of cores used. The tool is easy to install with PIPX and doesn't require any external dependencies that can't be installed through PIP.

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, there are unit tests for each query. Giving 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 a project.

7.1 Agda Tree Unit Tests

Each query and the options that the queries can take are unit tested. These test use the TypeTopology definition and module dependency graph. While testing every combination of input for this graph and checking the output isn't possible, the test instead evaluates one example of that query. For example, to test the dependencies query the definition `InfinitePigeon.Addition.n-plus-zero-equals-n` is passed in, and the output is checked given an expected response. This is done for each query, along with each of their options. So, 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 do not need to be tested for their integration as the CLI is based of the queries directly.

Is it necessary to say integration testing wasn't done

7.2 Agda Tree Performance

The performance of the queries has been measured, although, this will be highly dependent of the size of the graph and the Agda project. In this case TypeTopology was used, it is a large project with about 50,000 definitions. This gives a good representation of how long queries will take with most projects.

Should the milliseconds be formatted differently?

Most queries are completed in less 20 milliseconds. The exceptions being the cycles queries, the indirect uses query and path between a node or a leaf query. The cycles query takes about 622 ms to be completed, and the indirect uses takes about 3804 ms to be completed. These queries recursively call and traverse the whole graph multiple times, so this is expected. The path between two definitions queries 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 8 minutes or more, although, the command only runs once, after which, the graph is re-used.

7.3 Agda Comp Strategy Validation

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 section 5.2.1. 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 going through each step of the compilation order, if there are modules that are compiled in parallel then the dependencies are checked to be distinct. If so, they are removed from the graph and the next step is checked. By the end if all modules compiled in parallel had distinct dependencies then the overall compilation is safe.

The performance of Agda Comp is highly dependent on the Agda project, and will be assessed on section 9.2.2 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 negligible compared to the 5 minutes it normally takes TypeTopology to be compiled.

7.4 Conclusion

Agda Tree unit tests each query, checking if a certain input returns the correct output. Improving maintainability and stability of the code base. The tests are timed, with most queries being instant 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 validity of the strategies, whether the strategies were correct and safe. The Agda type checker doesn't give any perceivable error with unsafe or incomplete compilation, so it is critical to give a guarantee of a well done compilation. The time to create the compilation order was tested, but it is negligible compare to compilation time taking at most 50 milliseconds.

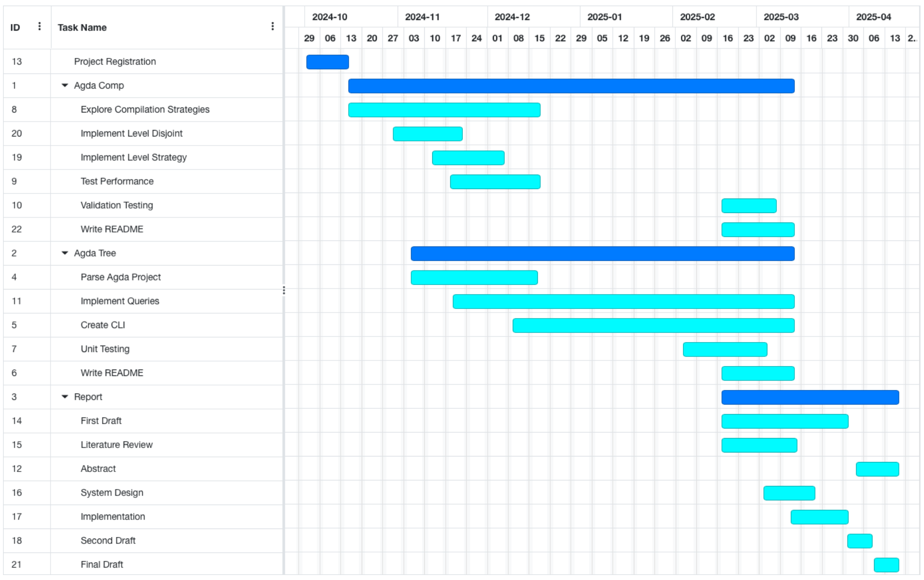
CHAPTER 8

Project Management

The timeline of the project can be seen in the below Gantt chart chapter 8. The focus on the first 2 months was to write Agda Tree and Agda Comp. For Agda Tree, the first milestone was to parse an Agda project and to extract the definition dependency graph. The next milestone is to implement the queries on the dependency graph. Once the queries were implemented, the last milestone was the CLI that gives the user a way to run those queries with their own parameters. Lastly, the queries were unit tested for correctness and for performance. This was 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 was also being developed. The first task of Agda Comp was to explore different compilation strategies and different methods to compile an Agda project. Once the strategies were selected, they were tested for correctness, safety and compilation time which was a major milestone. With the strategies implemented, the CLI was created that runs the strategy and compilation strategy. This was 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 improvements and additions for next week. This iterative approach allowed for a skeleton of the tools to be quickly created for my supervisor to test.



ID	Task Name
13	Project Registration
1	Agda Comp
8	Explore Compilation Strategies
20	Implement Level Disjoint
19	Implement Level Strategy
9	Test Performance
10	Validation Testing
22	Write README
2	Agda Tree
4	Parse Agda Project
11	Implement Queries
5	Create CLI
7	Unit Testing
6	Write README
3	Report
14	First Draft
15	Literature Review
12	Abstract
16	System Design
17	Implementation
18	Second Draft
21	Final Draft

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.

9.1 Agda Tree

Agda Tree extracts the definitions and relationships from an Agda project, and uses Agda to generate the module dependency graph then imports them into NetworkX. The queries outlined in table 4.2 for the definition graph and in table 4.3 for the module graph, are implemented. These methods are exposed through a CLI that the user uses to execute those queries on their Agda projects. The python library argparse validates the user input. Lastly, CLI the outputs the modules to console, which can be piped to other utilities. The CLI will work with any Agda project.

At first, HTML files that Agda generates were explored instead of s-expressions. These HTML files display and format the code with colours and links. Styling the text differently depending on if it was a definition, keyword, type, or operator. Also, the definitions were hyperlinks that linked back to the module that defined them. By exploring the HTML, a definition dependency graph could be created. The main issue becomes parsing the HTML files, finding which keywords fell into which definition is challenging. Switching to using s-expressions which are easier to parse was a significant step forward.

Clojure was considered instead of Python to store the graphs, as it is made to store information as graphs and make queries. However, Clojure requires Java, isn't popular, and it is more difficult to learn. This makes Python the better choice.

Agda Tree extracts s-expressions from an 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 7.2. Most queries are completed 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 well-documented in the `README.md` file and the `--help` option, making it easy to learn and understand. Since Agda is mainly developed on macOS and Linux, the tools were tested in WSL and a MacBook Air without issues. If the CLI is given bad inputs it returns an error from NetworkX explaining what the issue is. The code base is 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. The requirements as described on table 4.1 for functional requirements and table 4.4 for non-functional requirements.

9.1.1 Limitations

A 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 limits the usage of the query. Another limitation is the verbosity of the definition names and the amount of definitions. The TypeTopology definition graph has more than 50,000 definitions but in further inspection not all these definitions provide valuable information to the user. Sometimes they come from Agda's backend which isn't valuable. Also, the definition names become difficult to understand due to their length, and some definitions are ambiguous, so they require an ID number. Overall making the name of the definitions cumbersome to use.

9.2 Agda Comp

Agda Comp uses Agda to generate the module dependency graph and import it into Python. With this graph, Agda Comp lets the user pick between multiple strategies for compilation. The user chooses how many cores to use during compilation. Then the necessary index files and make file are generated, compiles the project and lastly cleans up the generated files. The CLI has a `README.md` and a usage message with the `--help` option making it user-friendly and easy to learn. The argparse library validates the user input to the CLI and gives a user-friendly error message if not. The tool detects the project structure from the user input, making it usable in any project structure. Therefore, Agda Comp meets its functional requirements, except it speeds up compilation for some Agda projects which is explored in section 9.2.2. The functional requirements were described in table 4.5.

Agda Comp works in any project, it has a simple CLI for choosing the compilation strategy. It was tested within WSL and macOS environment, for compatibility. The codebase is well-documented, and the strategies were tested for correctness and safety. Therefore, Agda Comp meets all of its non-functional requirements as described on table 4.6.

9.2.1 Limitations

This approach encounters two limitations, the first is that to create the module dependency graph it has to compile the project. Which means that for Agda Comp to save time the user only makes changes that do not alter the module dependency graph, such that a previous dependency graph can 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 type checking 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.

9.2.2 Compilation Strategies

The table 9.1 and table 9.2 show the compilation times of each strategy. The compilation testing consisted of 7 strategies across 3 libraries. The three libraries picked are TypeTopology [12], unimath [21] and Agda's stdlib [22]. These libraries vary in size and methodology, for example unimath structure is of many small independent modules while TypeTopology has less but longer modules. These libraries were chosen because they vary in size and strategy that explore the limitations of Agda Comp.

The 7 strategies tested are normal, which is the standard compilation that gives a baseline for the other strategies. 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 6.2.1, 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.

Should I add the tables here, or move some to the appendix?

Table 9.1: Results from WSL Testing Compilation Strategies

Time (s) % of normal	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 Mac Mini

Time (s) % of normal	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%)

Table 9.3: Results from MacBook Air M4 Compilation Strategies

Time (s) % of normal	Normal	Unsafe	Level A 2 cores	Level A 5 cores	Level B 2 cap	Level B 4 cap	Disjoint
TypeTopology	371 (100%)	234 (63%)	331 (89%)	298 (80%)	366 (99%)	334 (90%)	340 (92%)
stdlib	191 (100%)	134 (70%)	316 (165%)	267 (140%)	277 (145%)	273 (143%)	223 (117%)
Unimath	335 (100%)	295 (88%)	1437 (429%)	802 (239%)	674 (201%)	687 (205%)	314 (94%)

The tables show compilation time is affected on the project structure. The specs of the machines used can be seen in table A.1, table A.2 and table A.3. TypeTopology benefitted the most from the compilations strategies achieving about a 40% faster compilation in WSL, 23% faster in the Mac Mini and 20% faster in the MacBook Air. Although the results from the remaining Agda projects aren't as promising, they show improvement ranging from 21 % better to 329 % worse. 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 for all projects.

The cause in the difference in compilation improvement per project, is likely due to the limitation discussed in the section 9.2.1. Since unimath has many 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. This is a hypothesis that could be explored further.

9.3 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. 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 imports the module dependency graph from any Agda project and compiles the project in parallel by using different strategies. Although, the effectiveness of those strategies varies depending on the system and the Agda project. There is a limitation where creating the module dependency graph compiles the project. Meaning this tool is used when the user makes a change that doesn't alter the module dependency graph, limiting its scope.

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 the user 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, depending on the size of the Agda project. Queries can 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 for the user to compile their projects. Testing with these strategies using different parameters, across different Agda libraries, 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 is compiled 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 interact with the dependency graph could be a GUI where the user can save shorthands for relevant definition names. 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. 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. Also, to create the dependency graph the project is compiled which limits the usage of the tool, an option to create the dependency graph without type checking would help. Ideally this would be implemented as a part of Agda, but the source code includes import statements for every module so generating a module dependency graph from these import statements could be explored.

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] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[7] 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.

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

[9] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar. Common refactorings, a dependency graph and some code smells: an empirical study of java oss. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, page 288–296, New York, NY, USA, 2006. Association for Computing Machinery.

[10] 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.

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

[12] M. H. Escardó and contributors. TypeTopology. `https://github.com/martinescardo/TypeTopology`. Agda development.

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

[14] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[15] 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.

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

[17] Julian Vidal. AGDA HTML. `https://github.com/JulianVidal/AGDA_HTML`, Sept. 2025.

[18] 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.

[19] 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.

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

[21] E. Rijke, E. Stenholm, J. Prieto-Cubides, F. Bakke, and others. The agda-unimath library. `https://github.com/UniMath/agda-unimath/`.

[22] The Agda Community. Agda Standard Library. `https://github.com/agda/agda-stdlib`, Sept. 2024.

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

[24] 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.

[25] 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.

APPENDIX A

Appendix

Listing A.1: Agda Comp Help Message

```
usage: agda_comp [-h] [-c] [-j JOBS] [-s
    {level,levelb,normal,unsafe,disjoint}] module

Fast Agda type checker

positional arguments:
  module              Path to module to compile

options:
  -h, --help          show this help message and exit
  -c, --clean         Create dot file even if it already exists
  -j, --jobs JOBS     Cores that can be used
  -s, --strategy {level,levelb,normal,unsafe,disjoint}
                      The strategy that will determine the compilation
                          order, the
                      choices are: level: Sorts modules into levels,
                          each level
                      increses the length to a leaf, lvl 5 are the
                          modules that 5
                      modules away from a leaf. Each level is then split
                          into 4 files
                      or the value given by --jobs levelb: Sorts modules
                          by levels
                      like in 'level' but instead each level is
                          separated into n files
                      with --jobs modules each normal: Normal
                          compilation unsafe:
                      Tries to compile all the modules with 4 concurrent
                          index files
                      or --jobs files disjoint: Finds the biggest
                          modules that are
```

```
                        disjoint, if none are found the leaves are removed
                            then repeats
```

Listing A.2: Agda Tree Definition Help Message

```
usage: agda_tree definition [-h]
                         {create_tree,cycles,dependencies,
                         dependents, find,leafs,
                         module_dependencies,module_dependents,
                         module_path_to_leaf,nodes,path_between,
                         path_to_leaf, roots,save_tree,type,uses}
                         ...

positional arguments:
  {create_tree,cycles,dependencies,dependents,
  find,leafs,module_dependencies,module_dependents,
  module_path_to_leaf,nodes,path_between,path_to_leaf,
  roots,save_tree,type,uses}
    create_tree       Creates definition dependency tree from file,
                      -output option to set the path
                      to store the tree
    cycles            Cycles in graph
    dependencies      Definitions that definition d depends on,
                      -indirect will find the indirect dependencies
    dependents        Definitions that depend on definition d,
                      -indirect finds the indirect dependents
    find              Find definition through regex
    leafs             Definitions with no dependencies
    module_dependencies
                      Module dependencies of definition d,
                      -indirect finds the indirect
                      module dependencies
    module_dependents Modules that depend on definition d,
                      -indirect also gets the indirect
                      module dependents
    module_path_to_leaf
                      Longest path from definition d to any leaf only
                      counting modules
    nodes             List of definitions, if -c flag is set returns
                      the number of nodes
    path_between      Longest path between two definitions src
                      and dst
    path_to_leaf      Longest path from defintion d to any leaf
    roots             Definitions that aren't used
    save_tree         Save definition graph as pydot
    type              Types of definition d
    uses              Counts amount of uses per definition,
                      sorted in descending order, if -d is passed
                      in a definitino it will return the uses
                      of that definition

options:
  -h, --help          show this help message and exit
```

Listing A.3: Agda Tree Module Help Message

```
usage: agda_tree module [-h]
                        {complex_modules,create_tree,dependencies,
                        dependents,find,leafs,lvl_sort,nodes,
                        path_between,path_to_leaf,roots,topo_sort,
                        uses} ...

positional arguments:
  {complex_modules,create_tree,dependencies,
  dependents,find,leafs,lvl_sort,nodes,
  path_between,path_to_leaf,roots,topo_sort,
  uses}
    complex_modules   Get the top modules that have the most dependents
    create_tree       Creates modules dependency tree from file
    dependencies      Modules that module m imports
    dependents        Modules that import module m
    find              Find module through regex
    leafs             Modules with no imports
    lvl_sort          Level sort
    nodes             List of modules
    path_between      Longest path between two modules src and dst
    path_to_leaf      Longest path from module m to any leaf
    roots             Modules that aren't imported
    topo_sort         Topological sort
    uses              Counts how many times a module is imported, sorted
                      in descending order

options:
  -h, --help          show this help message and exit
```

Listing A.4: Level Sort Algorithm

```
def depth(g, node, mem):
    if node in mem:
        return mem

    children = list(g.successors(node))
    if len(children) == 0:
        mem[node] = 0
        return mem

    for c in children:
        m = depth(g, c, mem)
        mem |= m
        mem[node] = max(mem.get(node, 0), mem[c])
    mem[node] += 1

    return mem

def depths(g):
    m = {}
    for n in g.nodes:
        m |= depth(g, n, m)
    return m
```

Table A.1: Computer Specifications for WSL 13900hx

Specification	
Max Clock (GHz)	5.4
CPU P-cores	8
CPU E-Cores	16
RAM (GB)	24
Cooling	Active

Table A.2: Computer Specifications for Mac Mini M4

Specification	
Max Clock (GHz)	4.4
CPU P-cores	4
CPU E-Cores	6
RAM (GB)	24
Cooling	Active

Table A.3: Computer Specifications for MacBook Air M4

Specification	
Max Clock (GHz)	4.4
CPU P-cores	4
CPU E-Cores	6
RAM (GB)	16
Cooling	Passive