Tools for Generating and Exploring Agda's Definition and Module Dependency Graphs for Better Code Comprehension and Parallel Type-Checking

Undergraduate Final Year Project

Julian Camilo Vidal Polania

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

When dealing with large Agda projects it becomes difficult to navigate the project and slower to type-check. This paper addresses these issues with two CLI tools. Agda Tree which constructs dependency graphs from Agda definitions to help with code navigation and Agda Comp, which aims to optimize compilation time by parallelizing the type-checker using module dependency graphs. Agda Tree uses Andrej Bauer's s-expression extractor to generate the definition dependency graph and Agda's built-in command to generate the module dependency graph. The CLI lets developers query these relationships to better understand codebase. Agda Comp applies two strategies, level sort and disjoint level, to parallelize the module compilation while ensuring safety and correctness according to the module dependency graph.

The evaluation shows that Agda Tree extracts dependency graphs from Agda projects like TypeTopology, but the performance of the queries depends on the size of the project. Agda Comp achieves compilation improvements of up to 38% depending on the user's system, and project size and structure. However, the overhead of reloading Agda's interface files limits the gains in most scenarios.

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

1.1 Background

Large codebases with years of development can be overwhelming to navigate [6] and in Agda understanding the underlying definitions is important for new proofs. Dependency graphs can help in navigating the code base and identifying areas needing refactoring[9].

Agda is a functional programming language and proof-assistant based on propositions-as-types logic system [2]. Proofs are made by defining the most fundamental types and operations causing projects to rapidly expand. When making a proof, knowing how the types are fundamentally defined is helpful, but this information isn't readily available.

1.2 Motivation

Understanding relationships between definitions in large Agda projects is challenging due to their complexity. A tool that analyses these relationships and provides an intuitive interface can improve code comprehension and refactoring, reducing code decay [14]. For example, unused definitions can be removed, and highly dependent modules split into smaller parts.

Additionally, large Agda projects face long compilation times due to sequential type-checking of modules. Changes to high-level modules aren't a problem but low-level modules when modified will re-compile all their dependents. This issue worsens during refactoring, where structural changes lead to unnecessary re-checks.

Agda's sequential type-checking gives an opportunity for parallel compilation. Modules without dependency conflicts can be compiled concurrently, potentially reducing compilation time. A tool identifying safe modules for parallel compilation could lead to time savings.

1.3 Problem Statement

Agda doesn't have native commands for extracting project definitions. However, Andrej Bauer's extractor gives a view into Agda's internal representation as s-expressions [3]. This project aims to transform these s-expressions into dependency graphs for the user to explore for better codebase navigation.

To compile modules safely a module dependency graph is need which Agda already provides. The challenge is safely traversing the graph concurrently. Different strategies have to be tested against well-known libraries to gauge their effectiveness.

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 has access to queries that lets them 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.

CHAPTER 2

Literature Review

Common code representation graphs include Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG). These representations encode the behaviour of code and help with tasks like code vulnerability detection [15] and static analysis, where 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 the symbols in a project, this is gets 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

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 parallelise 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 optimise 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].

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

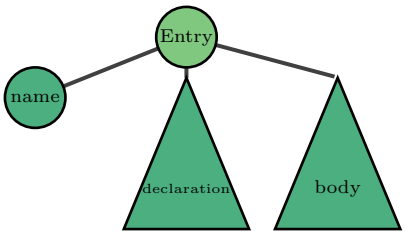
MLFMF is a collection of data sets for benchmarking systems that help mathematicians find relevant theorems when proving a new one [5]. The data sets are created from large libraries of formalized mathematics in Agda and Lean. They represent each library as a graph and as a list of s-expressions. Some 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

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 should be intuitive with minimal disruption to workflow. 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	Extracts the definition dependency graph from an Agda Project
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	Extracts the module dependency graph from an Agda Project
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 direct or indirect dependencies of a definition
2	Dependents	Get the direct or indirect dependents of a definition
3	Leafs	Gets the leaves of the dependency graph, which would be the definitions that have no dependencies
4	Module Dependencies	Gets the direct or indirect modules that a definition uses
5	Module Dependants	Gets the direct or indirectmodules that use a definition
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 direct or indirect dependencies of a module
2	Dependents	Get the direct or indirect dependents of a module
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 integrates seamlessly and works on a variety of projects regardless of size. To not disrupt the developer, the queries must respond in less than 1 second. 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 1 second
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 dependencies 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.

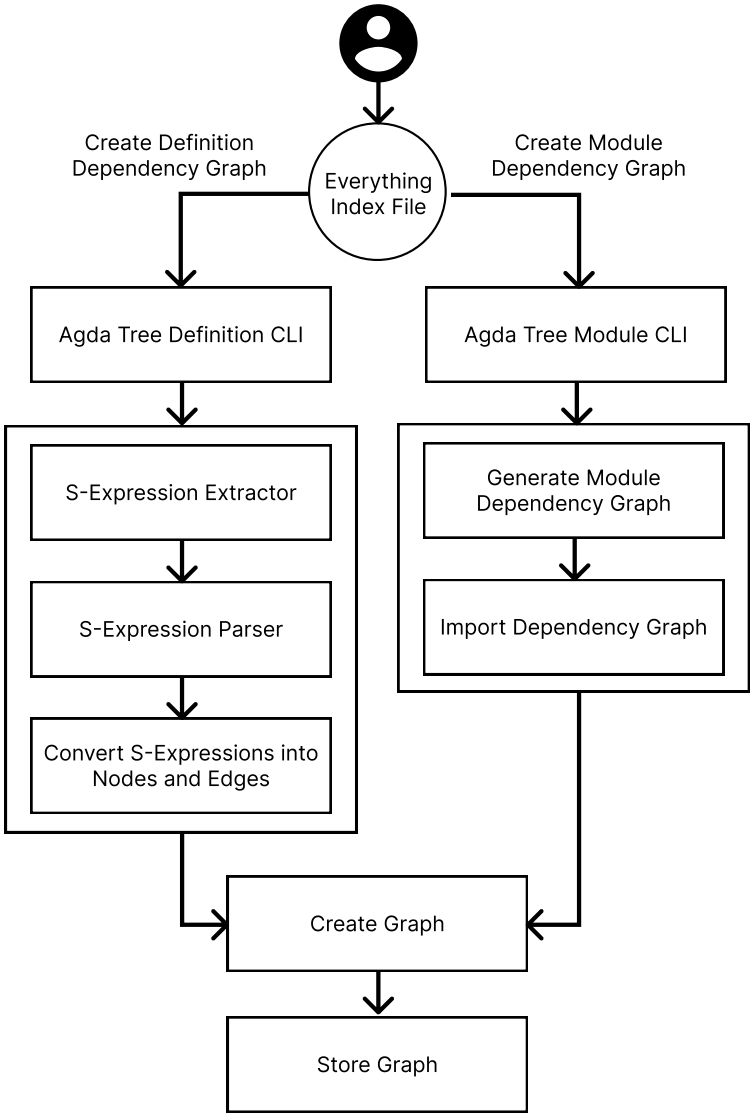


Figure 5.1: Agda Create Tree Diagram

fig. 5.2 is a Class Diagram, showing the classes and methods to operate the command-line. The CLI parses the input by the user and delegates 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.

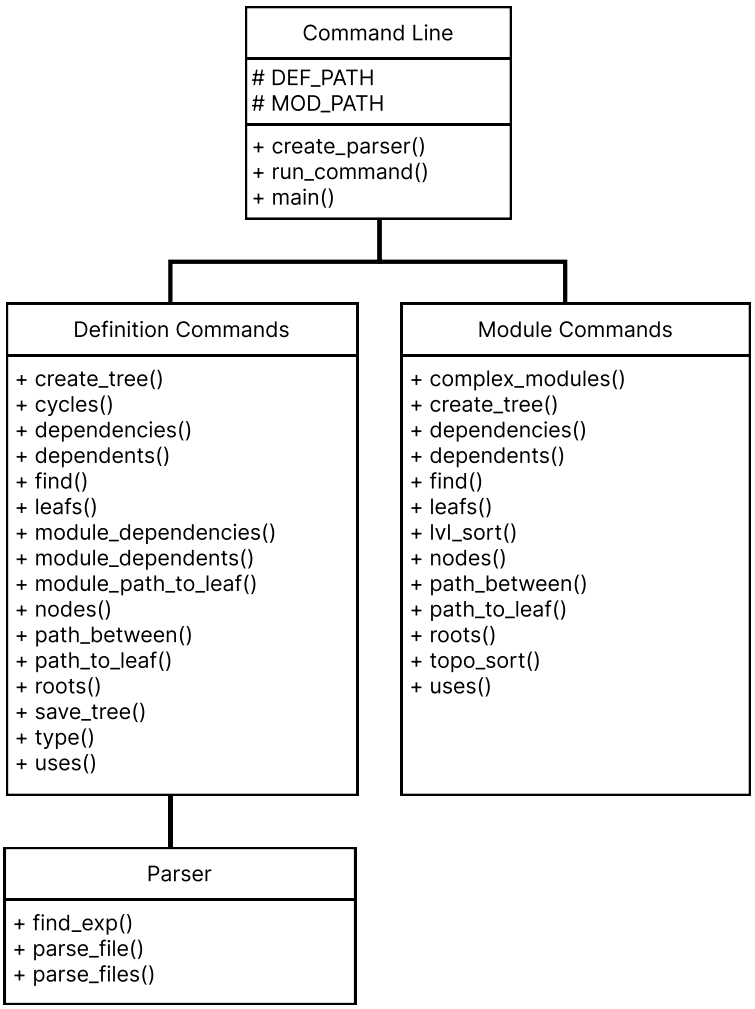


Figure 5.2: Agda Tree Class Diagram

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.

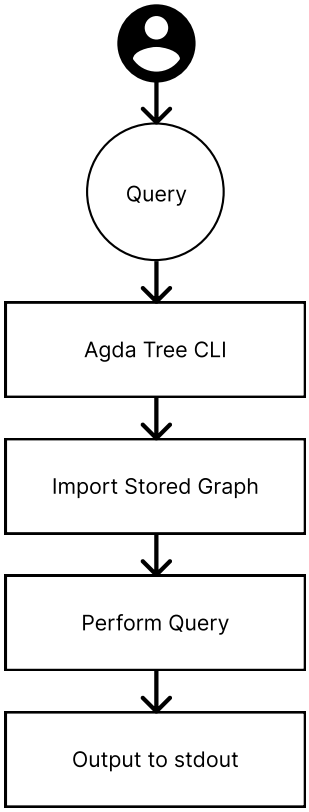


Figure 5.3: Agda Tree Query Diagram

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.

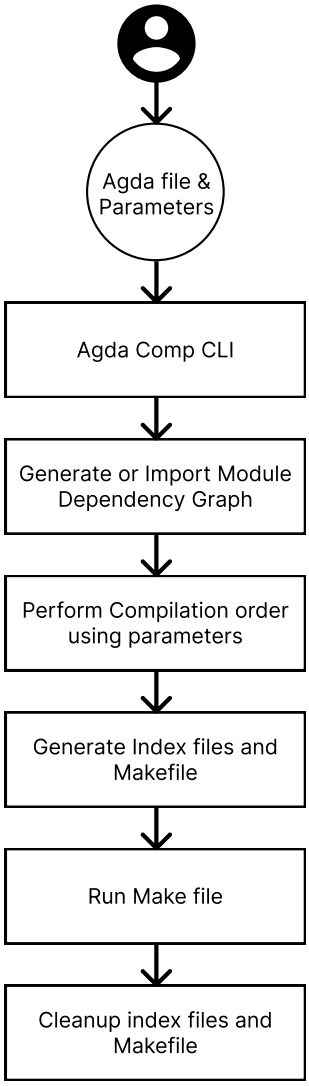
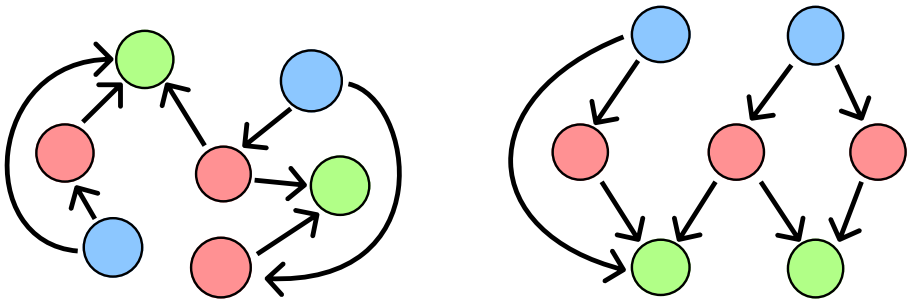


Figure 5.4: Agda Comp Diagram

5.2.1 Level Strategy

The level strategy groups modules by their maximum distance from leaf nodes, enabling parallel compilation within each level. Where leaf nodes at level 0 and Level n depends on modules $n-1$ or below. This algorithm is visualized in 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

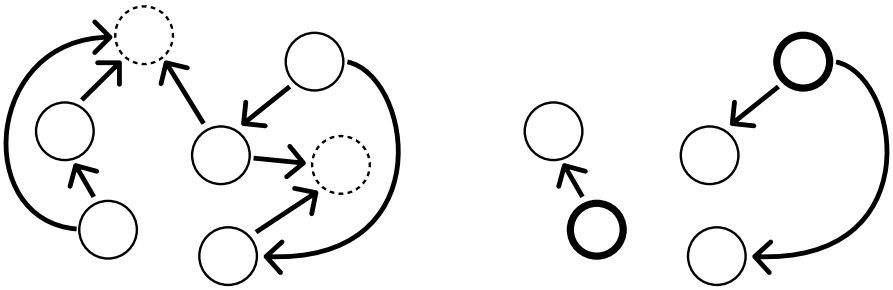
Figure 5.5: Level Strategy Example

This strategy involves sorting modules into levels and compiling each level sequentially, with concurrent compilation within levels which only depend on previous levels. It guarantees that all modules in the dependency graph are compiled, making the algorithm both safe and correct.

One advantage of this approach is it quickly generates the compilation order. However, the strategy is limited. If a level contains only a few modules, opportunities for parallel type-checking are low. Additionally, there are times when two modules with distinct dependencies could be compiled in parallel, leading to significant time savings with little overhead. However, this method would first compile all shared dependencies before compiling these two modules concurrently. This limitation becomes relevant during implementation (section 6.2.1), as the overhead associated with parallelization can outweigh its benefits.

5.2.2 Level Disjoint Strategy

The level disjoint strategy targets the weakness of the level sort strategy by finding modules with many dependencies that can be compiled in parallel. It identifies disjoint modules—those with distinct dependencies—and compiles them concurrently. If no such modules are found, leaf nodes (modules with no dependencies) are compiled first, and then it repeats. Once a module is compiled, it no longer considered. This approach reduces the number of modules to compile at each step and ensures all modules are eventually compiled. The is illustrated in 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 is both safe and correct, as it avoids compiling conflicting modules and guarantees that all modules in a project are compiled. It better manages parallelization overhead by targeting larger modules, reducing repeated loading of interface files during compilation. However, identifying disjoint modules is challenging for projects with many modules. A brute-force approach is impractical, so a greedy algorithm is used to approximate disjoint groups. This algorithm sorts modules based on their dependencies and groups them into buckets of disjoint modules for parallel compilation. This approach won't find the optimal solution and is further explained in section 6.2.1/

Despite its benefits, its benefit depends on project structure. Libraries like unimath [21], which consist of many smaller independent modules, benefit more than libraries like TypeTopology [12], which have larger connected modules.

5.3 Conclusion

The diagram in fig. 5.2 shows 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 code for this chapter can be found at [17]. Agda Tree creates a dependency definition graph from an Agda project and lets the user query this graph through a CLI. This system must meet the functional requirements described on table 4.1 and the non-functional requirements described on table 4.4. The CLI implements all the queries described on table 4.2 for the definition graph and the queries described in table 4.3 for the module graph. Agda Comp uses the module dependency graph to execute a strategy that returns the compilation order of the modules. It then compiles the project using that order. This system must meet the functional requirements described on 4.5 and the non-functional requirements described on 4.6.

6.1 Agda Tree

Agda Tree [17] 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 dependency graphs

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 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 6.1.3 to make sure the usability non-functional requirement is met from table 4.4.

S-expression parser

The S-expression parser extracts only the relevant information from Agda's s-expressions. Specifically the `:entry` and `:name` tags, which represent definitions and their dependencies. The s-expressions are generated and stored in a directory, then processed using Python's sexpdata library to load them into lists. These lists are traversed recursively to collect the required tags.

The parsing methodology involves finding all `:entry` tags, each representing a definition. For every `:entry`, the nested `:name` tags are extracted to construct a dictionary where each key is a definition name, and its value is a list of dependencies. This dictionary maps all definitions to their dependencies. A helper function, `find_exp`, is implemented to recursively traverse the s-expression structure and collect specific tags.

The parser processes each module file separately and in parallel, generating a dictionary for each. These dictionaries are then merged into a single dictionary containing all definitions and dependencies across the project. Additionally, a

similar process is applied to extract type information by analysing `:type` tags within the s-expressions. This produces another dictionary mapping definitions to their types.

Building definition graph

The parsed dictionary from section 6.1.1 constructs the definition dependency graph using NetworkX, where keys become nodes and dependency lists form edges. Nodes are added via `add_nodes_from()` using dictionary keys, while edges are created through `add_edges_from()` with `(definition, dependency)` tuples. This creates directed edges from definitions to their dependencies. The resulting graph is serialized using Python's Pickle library for storage and reuse.

Definition names originally include module paths and identifiers from Agda's internal representation. Before graph creation, these identifiers are removed unless needed to resolve naming conflicts improving readability.

Building module tree

Agda has a built-in feature that creates a DOT file of the module dependency graph of an Agda project. The file is imported into NetworkX, serialized and stored for future use.

6.1.2 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].

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 regex pattern. There is a `-name` option, if true then it will match the name of the definition otherwise matches the whole name including its modules.

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

Dependencies

The dependencies query gets all dependencies (direct or indirect) of a definition by traversing its child nodes in the directed graph.

```
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 all dependents (direct or indirect) of a definition by traversing its parent nodes in the directed graph.

```
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 they do not have children.

```
agda_tree definition leafs
```

Module Dependencies

The module dependencies query gets the modules of the definitions from the dependencies query. This query is exclusive to the definition graph.

```
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

The module dependents query gets the modules of the definitions from the dependencies query. This query is exclusive to the definition graph.

```
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. First all the simple paths are found, then they are measured for length and the largest one is returned.

The definition dependency graph is cyclic while the module dependency graph is acyclic, this causes a difference in performance. The definition graph also significantly large, so the amount of paths grows faster.

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

Roots

The roots query gets the definitions that have no dependents, meaning they do not have parents.

```
agda_tree definition roots
```

Use Count

The use count query gets the numbers of times a definition appear as a dependency (directly or indirectly). 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

The module path to leaf query gets the modules of the definitions from the path to leaf query. This query is exclusive to the definition graph.

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

Type

The type query gets the type of the definition. This data is collected during the building of the definition graph. This query is exclusive to the definition graph.

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

Cycles

The cycles query gets the simple cycles in the graph. Simple cycles are cycles where nodes aren't repeated. This query is exclusive to the definition graph.

```
agda_tree definition cycles
```

Save Tree

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

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

Path Between

The path between query finds the longest simple path between to definitions. Simple paths are paths that do not repeat nodes.

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

Level Sort

The level sort query sorts the modules into levels based on how far they are from a leaf module. This sorting is explained in section 5.2.1. This query is exclusive to the module graph.

```
agda_tree module lvl_sort
```

Topological Sort

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

```
agda_tree module topo_sort
```

6.1.3 Command-Line Interface

The CLI is implemented in Python, a widely used language that most users are familiar with, for easier modification by the user. Python's argparse library is used to handle user input, generate help messages, and manage command parsing. The CLI includes a main parser with two subparsers: one for definition queries and the other for module queries. Based on the user's input, the main parser delegates tasks to the appropriate subparser.

Query methods are stored in two files: `def_cmds.py` (for definition queries) and `mod_cmds.py` (for module queries). Using Python's inspect library, the CLI

automatically generates commands from these functions, extracting their names, parameters, and documentation. Positional parameters become required inputs, while optional parameters are specified with flags. This approach ensures that adding or modifying queries only requires changes to the respective function, with no need to manually update the CLI. Agda Tree's definition and module help messages can be seen below in listing 6.1 and listing 6.2 respectively.

Listing 6.1: 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 6.2: 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
```

Once user input is validated, the CLI loads the appropriate graph (definition or module) and executes the query. Using the query name the respective function is called. The query output is printed to the console, enabling integration with other terminal commands via piping.

6.1.4 Installation

To distribute and install the tool, it is packaged using a `pyproject.toml` file, which defines the project's metadata, dependencies, and supported Python version. The Hatchling backend is used to build the distribution, enabling installation on other systems.

For installation, Python's package manager, PIP, can be used to install the tool locally. Simply run `pip install .` in the project directory. However, it is recommended to use PIPX. PIPX isolates the tool's environment from the rest of the system, preventing dependency conflicts. Users can install PIPX through their respective package managers.

6.1.5 Installing Agda S-Expression Extractor

The Agda S-expression extractor, an extension of the Agda backend developed by Andrej Bauer [3], is not pre-built for download and must be manually built.

Agda Tree includes a script that automates this installation.

The script clones the repository into a temporary directory, checks out the branch compatible with the latest Agda version (2.7.0.1), and uses Stack to build the extractor. Stack, a Haskell build tool and dependency manager, is commonly used by Agda developers and can be installed via most package managers. The repository includes a YAML configuration file with instructions for building the binaries. Once built, the binary is copied to the `/.local/bin/` directory, which is typically included in the user's path.

After installation, the extractor can be accessed using the agdasexp command. The Agda Tree tool automatically detects this command and uses it to generate definition dependency graphs.

6.2 Agda Comp

Agda Comp[17] automatically creates the module dependency graph as described in section 6.1.1, using Agda's built-in command. The graph is passed into a strategy, that 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 processes the module dependency graph, represented as a directed graph using NetworkX. In this graph, nodes are modules, and edges points to its dependencies. To compile a module, all its dependencies must be compiled first. Additionally, two modules cannot be compiled simultaneously if they share dependencies, as this could lead to conflicts. A valid strategy compiles all modules safely.

The strategies produce a 3D array that defines the order of compilation and identifies which modules can be compiled in parallel. An index file groups modules that are compiled together. If two indices have disjoint dependencies, they can be compiled in parallel. The array specifies the compilation order by grouping indices into sequential phases. For example:

```
[
  [[module1, module2], [module3, module4]], # Phase 0
  [[module5]],                               # Phase 1
  [[module6, module7]]                       # Phase 2
]
```

In this example, `module1` and `module2` are compiled together in one index file, while `module3` and `module4` are grouped into another index file. Both index files are compiled in parallel during Phase 0. In Phase 1, `module5` is compiled on its own, followed by `module6` and `module7` in Phase 2.

When Agda compiles a module, it loads the interface files of its dependencies. By grouping multiple modules into a single index file, shared interface files are loaded once for the entire group rather than repeatedly for each module.

Finally, the make generator subsystem uses this array to create both index files and a Makefile. The Makefile ensures that indices are compiled in the correct order while taking advantage of parallelism wherever possible.

Level Strategy

The level strategy, as explained in section 5.2.1, orders modules into levels based on their maximum distance from a leaf node. Dependencies in previous the levels are compiled before modules in the next levels. This guarantees safe parallel compilation within each level. The algorithm works recursively as shown below in listing 6.3. Once sorted, modules at the same level are grouped into index files, where each index file can be compiled concurrently. This array is then used to generate a Makefile that runs the compilation sequence, as explained in section 6.2.2.

Listing 6.3: 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
```

When Agda type-checks a module, it reloads the dependency's interface files which could be large. To avoid reloading these files, the strategy groups modules within the same level into index files. Then interface files will load once per index rather than per module, reducing overhead. Two methods are employed to split modules into index files:

Method A: modules are split evenly across index files, where the split is guided by the available CPU cores. Creating more index files that cores could cause two compilations to share a core, reducing effectiveness.

Method B: caps the number of modules per index file, creating more files when the threshold is exceeded. While this may produce more index files than cores, it could mean that while one compilation is waiting for their interface files another is running their compilation.

These approaches (labelled Level A and Level B in section 9.2.2) were tested to evaluate their impact on compilation time across multiple projects and computers.

Level Disjoint Strategy

The level disjoint strategy, as described in section 5.2.2, addresses the limitations of the level sort strategy by focusing on large modules with many distinct dependencies. These modules, referred to as disjoint modules, are compiled first. If no disjoint modules are found, the strategy compiles leaf nodes, eliminating shared dependencies. This iterative approach means that the algorithm terminates when all modules have been compiled.

Identifying disjoint modules is challenging, especially for large projects with hundreds of modules. A brute-force approach is computationally not possible. So a greedy approach is used to approximate these modules. The algorithm sorts modules: first by the number leaf nodes they depend on (in ascending order), and then by the total number of dependencies (in descending order to break ties). This sorting prioritizes modules with few leafs and many dependencies.

After sorting, the algorithm groups modules into buckets based on their leafs. A new bucket is created for the first module in the sorted list, and subsequent modules are either added to existing buckets or ignored based on how their leafs overlap. Modules that share all leaf dependencies with a bucket are added to that bucket, while those sharing only some dependencies are ignored. Buckets represent groups of modules that can be compiled in parallel safely.

The compilation order array generated by this strategy begins with a step for compiling the leaf nodes sequentially. The buckets are then added as parallel compilation step. If only one bucket is found it is combined with leaf nodes into a single compilation step to avoid overhead. This ensures that all modules are eventually included in either a leaf step or a bucket step.

6.2.2 Building The Make File And Indices

An index file is a module that imports other modules but contains no definitions. Its purpose is to allow the type-checking of multiple modules as a group. Each index file starts with flags that modify the behaviour of the type checker. These flags vary between projects and are extracted from an index file provided by the user. Below is an example of an index file from the TypeTopology project:

```
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}
```

Index files are named according to their compilation order. For instance, `index-0-0` and `index-0-1` represent files compiled first in parallel, followed by `index-1-x` files compiled after.

The Makefile defines the list of commands required to compile a project. It consists of "targets," which specify what to build, how to build it, and

their prerequisites. For example, to compile `index-1-2.lagda`, its prerequisites (`index-1-0.lagda` and `index-1-1.lagda`) must be compiled first. The `all:` target compiles the entire project starting from the highest-level index file (`index-2-0.lagda`). Agda compiles modules into interface files with the `.agdai` extension. If an interface file does not exist, it triggers its compilation before dependent targets.

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
```

If two targets do not share dependencies, they are compiled in parallel automatically by the Makefile. For instance, `index-1-0.lagda` and `index-1-1.lagda` can be compiled together.

The Makefile is generated based on the output of the chosen compilation strategy. Each index file corresponds to a target named after its position in the compilation order array, with prerequisites being the previous index files in the array. The final target includes all previous indices and is added to the `all:` target for complete project compilation. Initial index files without dependencies have no prerequisites.

6.2.3 Command-Line Interface

The Command-Line Interface (CLI) is implemented in Python, as most users are likely to have it installed and may already be familiar with it. Similar to the Agda Tree CLI (section 6.1.3), the argparse library is used to parse user input, validate commands, and generate usage and help messages. The parser accepts the path to the module file to compile as a positional parameter while allowing additional options to customize the compilation process, such as selecting a strategy. The help message detailing these options is displayed in listing 6.4 below.

Listing 6.4: 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
```

The CLI provides three main options: `clean`, `jobs`, and `strategy`. By default, the tool reuses the existing module dependency graph and excludes already compiled modules from further compilation. The clean option forces regeneration of the module dependency graph and deletes the `_build` directory containing compiled modules. Agda Comp determines whether a module is already compiled by checking if its corresponding interface file exists and whether it has been modified since its creation. If a module has been altered, it is recompiled. The jobs option specifies how many CPU cores should be used during compilation. Finally, the strategy option allows users to choose which compilation strategy to apply.

When the user provides the index file path and options, the tool generates index files and a Makefile. These files are then moved to their respective directories: index files to the project source directory and Makefiles to the project root directory. The command `mk compilation.mk` is executed, where `compilation.mk` is the generated Makefile. This command compiles the project using the specified number of cores. Upon completion, both index files and Makefiles are deleted to maintain a clean project directory. If the user cancels compilation midway, a listener will remove the temporary files, preventing clutter in the project.

For example, to compile TypeTopology using two cores, deleting previously compiled modules, regenerating the module dependency graph, and applying the Level B strategy, the following command can be used:

```
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 6.1.4. 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 performance.

For Agda Comp, testing focused on validating the correctness and safety of its compilation strategies. The tests ensured that all modules in a project were compiled, no modules were skipped, and no module was compiled simultaneously with one of its dependencies. Additionally, the time required to generate the compilation order was measured but negligible compared to the overall project compilation time.

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.

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 Type-

Topology 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.

Most queries are completed in under 20ms (milliseconds), except cycles queries, the indirect uses query and path queries. The cycles query takes about 622ms to be completed, and the indirect uses takes about 3804ms 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 indefinite 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 project timeline is shown in the Gantt chart (chapter 8). During the first two months, the focus was on developing Agda Tree and Agda Comp. For Agda Tree, the initial milestone was parsing an Agda project to extract the definition dependency graph. The next step involved implementing queries for this graph. Once the queries were functional, the final milestone was creating the CLI that allowed users to execute these queries. The queries were then unit tested for correctness and performance and a stable version of the CLI tools was finished by the demonstration date.

At the same, Agda Comp was being developed. The first task was exploring various compilation strategies and methods to compile an Agda project. After selecting the strategies, they were tested for correctness, safety, and compilation time, marking a major milestone. Following this, a CLI was implemented to execute these strategies, and this was also completed by the demonstration date.

An agile software development methodology was used throughout the project. As shown in the timeline, multiple tasks ran concurrently until they were completed for the demonstration. Weekly meetings with my supervisor facilitated iterative progress, where I presented a working state of the project along with a summary of achievements since the previous meeting. My supervisor provided feedback and suggestions for improvements or additions for the following week. This iterative approach enabled rapid prototyping of tools that my supervisor could test early in development.

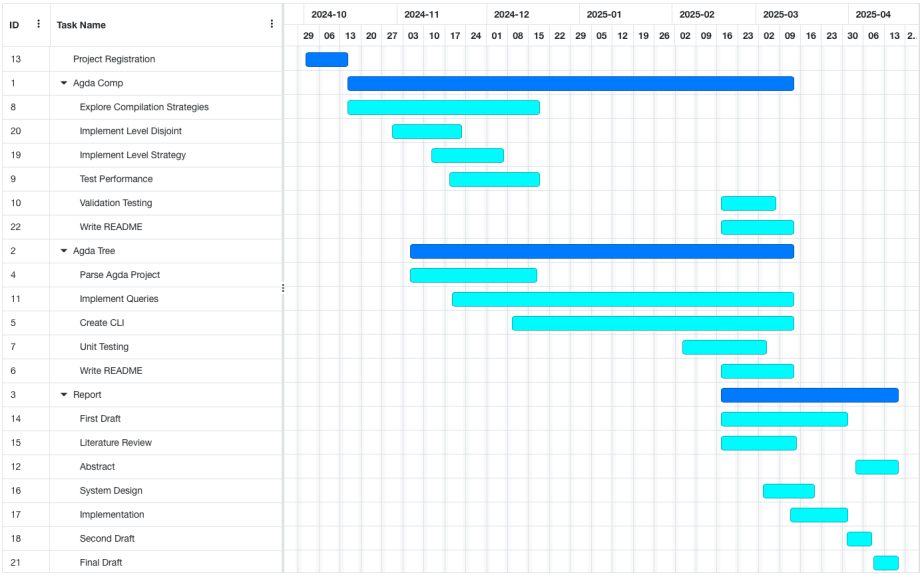


Figure 8.1: Gantt Chart

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, the HTML files that Agda generates natively 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 parallelised 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.

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 M4 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%)

Table 9.4: Computer Specifications for WSL 13900hx

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

Table 9.5: Computer Specifications for Mac Mini M4

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

Table 9.6: Computer Specifications for MacBook Air M4

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

The specs of the machines used can be seen in table 9.4, table 9.5 and table 9.6. TypeTopology benefitted the most from the compilations strategies achieving up to 38% faster compilation in WSL. Although the results from the other Agda projects aren't promising, they show improvement ranging from 21 % better to 329 % worse. The compilation times appear to depend heavily on the user's system. Also, the unsafe strategy shows improvements in all tests which shows potential for time savings.

The difference in compilation time between projects, 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 reloaded. 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. Lastly, the unsafe strategy showed promise of possible time savings.

CHAPTER 10

Conclusion

Agda Tree extracts definition dependency graphs by using Andrej Bauer's s-expression extractor [3] and imports them into NetworkX alongside Agda's module dependency graph. The CLI provides users with queries to explore these graphs, helping understand codebase. The tool is user-friendly and straightforward to install, making it accessible to developers. However, its is somewhat limited by the size of the Agda project; queries can be slow for larger projects, and additional information from the Agda backend can clutter the CLI output. Despite these limitations, Agda Tree achieves its main goal of helping users navigate large Agda codebases effectively.

Agda Comp investigates two parallel compilation strategies: level sort, which organizes modules into levels for concurrent compilation, and disjoint, which identifies large independent modules for parallel processing. Both strategies were implemented into a CLI tool that allows users to compile their projects. Testing showed that TypeTopology benefited significantly from these strategies, achieving notable speed-ups, while other libraries experienced modest or no improvements.

Agda Comp faces two key limitations. First, calling Agda's type checker multiple times results in redundant loading of interface files, introducing overhead. Second, generating the module dependency graph requires compiling the project, limiting the tool's usability to scenarios where the graph remains unchanged. Consequently, while Agda Comp partially meets its goal of improving compilation times, its effectiveness varies depending on the system and project.

10.1 Future work

Definition names are verbose and unintuitive, future improvements could include a GUI to interact with the dependency graph 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.

In Agda Comp's current state it doesn't always improve compilation times. To address its limitations, the Agda type checker itself needs to be parallelised from within to avoid the overhead of reloading interface files. The other limitation could be avoided by generating the module dependency graph without type-checking it, this could be implemented by scraping the import statements from the source code.

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.