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

Dissertation

Julian Camilo Vidal Polania

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

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

I will mention:

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

Acknowledgements

Abbreviations

ACB	Apple Banana Carrot

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Add more citations to articles and different sources

Add section about s-expressions

1.1 Background

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

Agda is a functional programming language that can be used as a proof assistant, following the propositions as types logic system [3]. Since proofs in Agda can be created almost entirely from scratch, defining the most basic and fundamental types and operations it means that projects will quickly balloon in size. While most of the time it is possible to use higher level definitions when developing a proof, knowing the how the used types are fundamentally defined can be quite beneficial.

1.2 Motivation

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

Also, large Agda projects tend to deal with long compilation times. Agda is a proof assistant, during ”compilation”, it type checks the entire project which can take a 5 minutes or more depending on the computer and project. Agda will keep track of what modules have already been compiled, so working on high level modules isn’t an issue as Agda will only type check files that have changed in the project and high level modules don't have dependents that would need to be re-checked. In contrast, low level files with many dependents, such as a module defining the natural numbers, will cause a significant portion of the project to re-compile. This effect will be particularly pronounced during refactoring where significant changes will be made to the bottom of the project, causing Agda to re-check the whole project making most of the time spent refactoring, spent waiting.

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

1.3 Problem Statement

Agda doesn't provide a built-in tool to extract the definitions from a project, but there is an s-expression extractor Agda version by Andrej Bauer [4] that will allows a better view into Agda's internal representation. S-expressions aren't well suited data structure for the queries on definition relationships, mostly contain syntax information that can be discarded and each module has an s-expressions file. A data structure that lends itself well towards queries of relationship between definitions, that only contains information about the definitions and stores all the definitions from a project, such as a graph. A tool that could explore the dependency graph of the definitions, would allow for a faster understanding of a large code base.

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

1.4 Objectives

This objective of this project is to create a dependency graph of defintions from an Agda project by using the s-expression representation of modules using the s-expression extractor [4]. Then create build a (Command Line Interface) CLI that will create the dependency graph from any Agda project and give the user access to commands that query the dependency graph. These queries will let the user explore the graph, in order to gain insight into the Agda project. As there aren't currently any tools that allows you to query the definitions of an Agda project.

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

1.5 Project Structure

First, there will be an explanation of what s-expressions are and how they are being read. These s-expressions use tags to name different symbols in a module, only the tags that contain definitions are important. Then, show how the s-expressions are structured, the way that defintions can be found and getting their dependencies. Using the information gained for the s-expressions, they will be turned into a graph that can be easily queried for relationships. Some example queries will be demonstrated, to underline their utility and implementation.

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

CHAPTER 2

Literature Review

Add literature about code decay

There are many ways to represent code using graphs, each with their own utility and purpose. There are Abstract Syntax Tree (AST), Control Flow Graph (CFG), Data Flow Graph (DFG), Program Dependence Graph (PDG), and Code Property Graph (CPG). These represenations encode all the behaviour and properties of a project, they can have many uses such as for code vulnerability detection [10]. These representation also allows for static analysis, where code isn't ran but the structure is analyzed for software validation [9]. For this project, these graphs encode far more information that is require, as for the dependency graph only the relationship between definitions is of value.

Add section explaining dependency graph

2.1 Related works

The Language Server Protocol (LSP) is used by IDEs, such as Vistual Studio Code and Intellij, to provide features like goto defintion and goto references. The LSP sits between the code editor and the language server, it is the language server that analyzes the code structure to support the features [11]. Language server has the functionality to implement this project, but, they are meant to be used while editing code and the language server isn't made to be used with custom queries. Examples of LSP's are Jedi [14] and Agda's langauge server [2], while they contain the functionality needed they are only meant to work on the current open file, not a whole project.

Ctags [15] is a tool that indexes all the symbols in a project, this is helpful to get all the definitions from an Agda project. But it doesn't capture the relationship between the symbols it finds.

Graph Buddy is an interactive code dependency browsing and visualization tool [7]. It takes large Java codebases turns them into Semantic Code Graphs (SCG), and creates a visualization of this graph. This graphs shows dependencies between modules, classes and methods which helps the developer better understand the project and tackle the pervasive issue of code decay [5]. Graph

Buddy is integrated as to an IDE as a plugin, where the user can seamlessly explroe the visualization.

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

2.2 Parallel Compilation

Add section explaining DAGs

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

An example is the work by Newton et al [13] which seeks to parallelize type-checking with Haskell. Also, the work by Zwaan et al. [17] using scope graphs for incremental type-checking.

However, this project doesn't aim to optimize the type checking algorithm themselves, rather, find independent modules that can be type checked together. This is aligns more close with the following paper that explores reducing FPGA compile time by changing from a monolithic compilation style to compiling separate blocks in parallel [16].

Fundamentally the compilation problem in this project is a scheduling problem, which is NP-complete[12]. This means that there many algorithms that attempt to tackle this problem by following different assumptions as shown by Yves Robert [12].

Add section explaining scheduling problem

Add section explaining s-expressions

2.3 Conclusion

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

Slow compilers is a common problem, hurting developer experience. Due to the monolithic nature of compilers, parallelization becomes a route to follow when optimizing compilation time. While parallelization can be applied to the type-checking algorithm itself, this project looks to type check modules in parallel while the type checking algorithm remains the same. This problem is closer to a scheduling problem, where the goal is to find the optimal way to assign tasks to multiple machines to reduce completion time. There is no tool in Agda that attempts to apply an scheduling algorithm to the type-checking of modules, which could lead to significant speed ups.

CHAPTER 3

Legal, Social, Ethical and Professional Issues

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

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

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

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

CHAPTER 4

System Requirements

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

4.1 Agda Tree

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

4.1.1 Functional Requirements

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

Table 4.1: Agda Tree Functional Requirements

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

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

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

Table 4.2: Agda Tree Definition Queries

ID	Name	Description
1	Dependencies	Get the dependencies of a definition 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 leafs 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 uses a definition both directly and indirectly
6	Path to Leaf	The longest path from a definition to any leaf
7	Module Path to Leaf	The longest path from a definition to any leaf but only following the modules of the path
8	Roots	The definitions with no dependents, meaning they aren't used anywhere
9	Definition Type	The definitions used for the type of the definition
10	Use count	Counts how many times a definition is used
11	Cycles	Returns the cycles in the graph
12	Save Tree	Saves the tree into a dot file
13	Path Between	Finds the longest path between two definitions

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

Table 4.3: Agda Tree Module Queries

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

4.1.2 Non-Functional Requirements

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

Table 4.4: Agda Tree Non-Functional Requirements

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

4.2 Agda Comp

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

4.2.1 Functional Requirements

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

Table 4.5: Agda Comp Functional Requirements

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

4.3 Non-Functional Requirements

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

Table 4.6: Agda Tree Non-Functional Requirements

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

4.4 Conclusion

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

The functional requirements for Agda Comp show the features needed for the tool to be used, the tool must work with any Agda project and give the user choice on how to run their 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 neccessary and do so without compiling two modules at the same time which could cause concurrency issues. Since the time it takes to compile varies on the project size, performance measures aren't going to be made.

CHAPTER 5

Design

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

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

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

5.1 Agda Tree

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

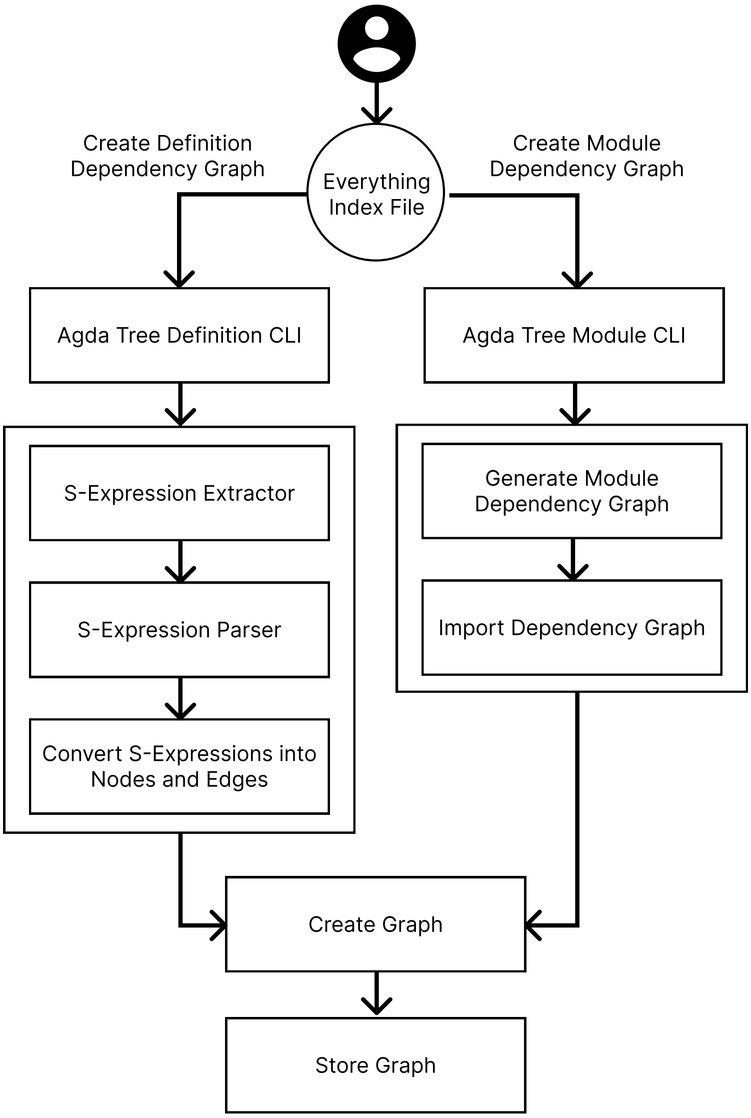

Figure 5.1: Agda Create Tree Diagram

Figure 5.1 is a Class Diagram, showing the classes and methods needed to operate the command line. The program starts with the command line entry point, that will parse the input by the user and delegate the running of the commands to the respective dependency graph. The command line stores the default path to the dependency graph, the definition and module dependency graph will have their own commands that can be run on them. These queries can be found in Table **??** for the definition graph and in Table **??** for the module graph for the module graph.

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

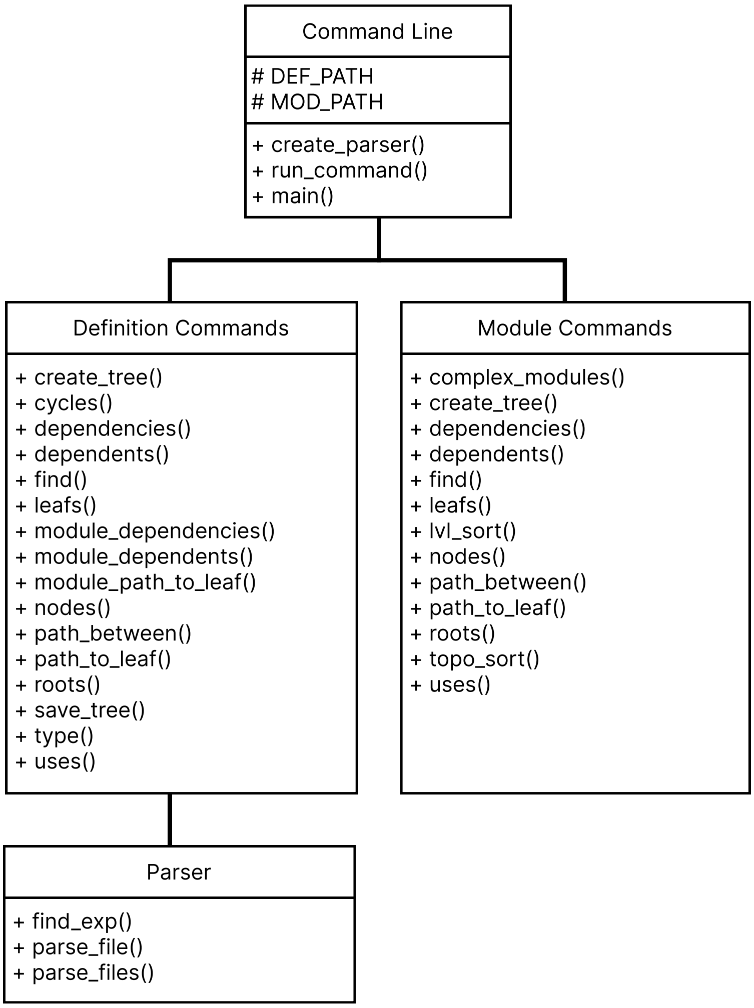

Figure 5.2: Agda Tree Class Diagram

Figure 5.1 shows how the CLI splits into two, depending on what dependency graph is being queried. When a user makes a query, they will select the dependency graph and the respective methods will perform that query. The output will be displayed in stdoutput, this makes sure that the output can then be used with other commands like piping and xargs.

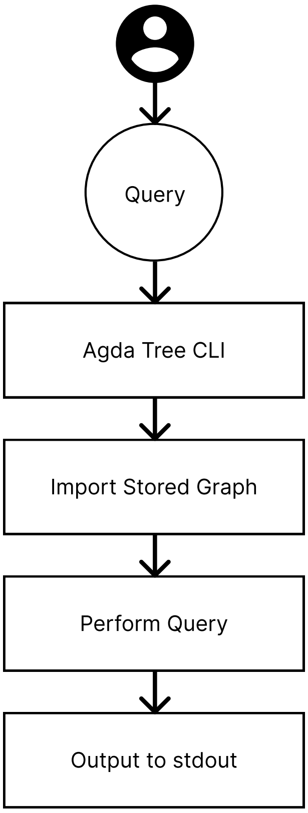

Figure 5.3: Agda Tree Query Diagram

5.2 Agda Comp

Figure 5.2 demonstrates how the user will interact with the Agda Tool. The user provides the module that will be compiled, next to some parameters defining the amount of cores used in the parallelization and what compilation strategy to use.

Add more stuff about Agda Comp, I don't know what

What are the algorithms that are going to be sued for Agda comp

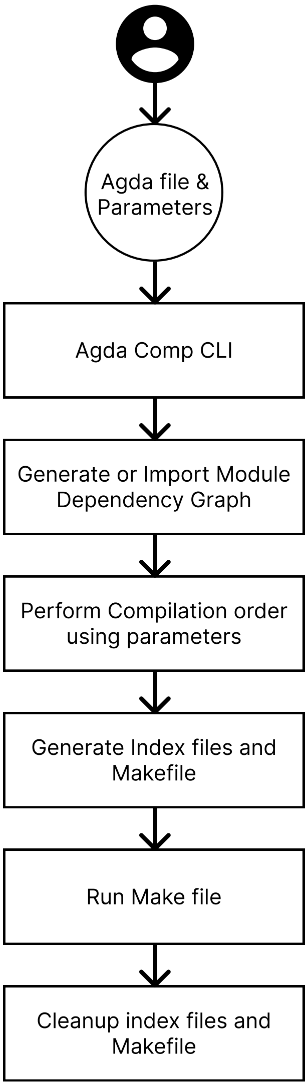
Agda file & Parameters
Agda Comp CLI
Generate or Import Module Dependency Graph
Perform Compilation order using parameters
Generate Index files and Makefile
Run Make file
Cleanup index files and Makefile

Figure 5.4: Agda Comp Diagram

5.3 Conclusion

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

The functionality of Agda Comp is modelled in Figure 5.2, this structure allows the user to select what compilation strategy to use and how many cores the compilation can use. Agda Comp is meant to allow the user to easily apply the compilation strategies that are going to be explored in chapter 6.

CHAPTER 6

Implementation

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

- File structure and project structure

6.1 Agda Tree

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

6.1.1 Building the definition tree

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

The solution is not to use the HTML files but use s-expressions. The s-expressions are the same way the HTML files are with the same information, except s-expressions are easier to parse. However, Agda doesn't natively generate s-expressions but Andrej Bauer, Matej Petković, Ljupčo Todorovski in their paper "MLFMF: Data Sets for Machine Learning for Mathematical Formalization" [6] created an s-expression extractor. The s-expression extractor is in the Agda backend and it will convert Agda files into s-expressions [4].

S-expression extractor

S-expressions are a notation that is used in Lisp programming languges, it represents programs and data as tree-like data structures [1]. 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 [6].

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

Table 6.1: Relevant S-expressions

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

Note that the s-expression extractor is a special version of Agda with an extended back-end, which means the user has to compile this Agda version and add it to their path. This will be handled by the cli described section 6.1.4 to make sures the usability non-fuction requirement is met from Table 4.4.

S-expression parser

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

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

The s-expressions are compiled into a directory, this project is implemented in Python so the library sexpdata will be used to load the raw s-expression files into Python lists. Theses lists will be analyzed recursively, finding the relevant tags. The strategy is to find all the :entry tags in the file, each :entry tag represents a definition as described in Table 6.1. For each :entry tag, find all the :name tags contained inside. With this information create a dictionary were the key is the definition name and the value will be a list of the :name tags. Since :entry tags describe each definition and the :name tags describe what definition is being used, the resulting dictionary will contain all the definitions along with their dependencies. To find the necessary tags a find_exp function is implemented that recursively finds all occurences of a given tag within an s-expression.

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

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

- How the s-expression are extracted
- How the extractor is installed
- how the s-expressions are parsed
- What are s-expressions and how they work
- How to get the s-expression definition and relationships
- Explains how the tag works in this context i.e. name are definitions
- How they are imported into networkx and became a tree
- Explain how it is stored pickled
- Handling issues with recursion
- Issue with the naming of the defitions
- issue with where clause

Building definition graph

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

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

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

6.1.2 Building module tree

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

- Explain what dot files are
- Explain how agda extracts it
- Explain how it is imported into networkx
- Explain how it is stored pickled

6.1.3 Queries

Most queries are defined similarly between the definition graph and the module graph but the module graph being acyclic means that it has different properties.

Find

The nodes query gets all the nodes in the graph, it returns a list with all the definition names.

Find

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

Dependencies

The dependencies query gets all the dependencies of a definition. Meaning what theorems does the definition need to be defined, either due to its type or what it uses to be proved true. Since this is a directed dependency graph and the definition's edges point towards its dependencies, the dependencies are the children of the definition. NetworkX provides a method for this: `graph.successors(definition)`.

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

Dependents

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

Leafs

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

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 definition was in. Although, this will cause repetitions from multiple definitions in the same module so they are added to a set to remove repetitions.

Module Dependants

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

Path To Leaf

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

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

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 leafs but instead of checking for outwards edges, check for inward edges with the command: `graph.in_degrees(node)` if it has none then its a root.

Use Count

The use count query gets the amount 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 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.

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.

Definition Type

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

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)`.

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)` .

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 don't repeat nodes. After finding all the paths, it measures their lengths and returns the maximum length.

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.

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.

- Explain how each query is implemented
- Explain what the english definition means for the graph
- Explain what the technical challenges could be
- Explain the algorithms that were used
- Explain the properties of each graph and how that limits the queries
- Explain the limitations of the algorithms used if any
- Design on how the queries are executed using argparse

6.1.4 Command Line Interface

The command line interface(CLI) will created using Python, as it is a popular language that most users will already have installed and most users will have some experience with. This makes it easier for users to add their own queries and make the changes they want. At first Clojure was considered to create this tool, as it is made to store graphs and make queries about them. However, Clojure requires Java, isn't popular and more difficult to create queries in. Also, NetworkX is in Python which would cause further issues. This makes Python the best choice.

Python includes a library called argparse to create command line interfaces. In Python the values passed in to a program are sotred ins sys.argv, argparse will parse sys.argv based on the parsers defined. argparse will also create help and usage messages to help the user.

There is a main parser and 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 should be used. Then the contrl is handled to the respective sub parsers.

The sub parsers are generated automatically from the methods that describe the queries. There are two files def_cmds and mod_cmds which store the functions that perform the queries. The functions in these 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 created is by first getting all the functions names which will be the queries inside the cmd file. For each function, the parameters it requires will be the input the user has to give. If it is a position parameter then the user must give it, otherwise, it is an optional parameter that can be given by the user with -optionName. 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.

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 -h
usage: agda\_tree definition [-h]
                         {dependencies}

positional arguments:
    dependencies      Definitions that definition d depends on,
        -indirect will find
                      the indirect dependencies
options:
  -h, --help          show this help message and exit


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 input given by the user is evaluated and the query can be run. To run the query the firs the appropriate graph is loaded depending on the user's choice, this must be created before. Since the name of the query is the same name as the function and Python allows for a function to be called with the name of the function as a string then the name of the query is used to find and run the function. The parameters of the function is the same as the input parsed from the user, so this can be passed in directly to the function. The output of the function is then printed to standard output.

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

- Explain why using python
- Explain what argparse is
- Explain how the functions are stored in a file and the methods are read into for extensibility, one responsibility principle and open close principle.
- How the function parameters are added to the cli
- Explain why it is good to be a cli tool, as it can be piped and used like any other command (wz, fzf, cp)
- How clojure failed
- How cycles are difficult andc ant find distance to leafe

6.1.5 Installation

- Explain how dependencies are handle
- Explain how this can be installed as a project using pip
- The tool automatically isntalls agdasexp

6.2 Agda Comp

6.2.1 Strategies

- Explain each strategy
- How it works
- What the motivation for it is
- Implication on parallelization
- How it was tested for safety and correctness
- Why using index files, to gropu these modules
- How the algorithm is safe and correct
- How make files work
- Why use make files over other options
- What the output of the algorithms is

- How the output is used.
- Difficulty with diferenct directory names and index flags
- The limitations of each algorithm, pros and cons
- How they deal with multiple projects

6.2.2 Command Line Interface

- Explain why using python
- Explain what argparse is
- Explain how the functions are stored in a file and the methods are read into for extensibility, one responsibility principle and open close principle.
- How the function parameters are added to the cli
- Explain how dependencies are handle
- Explain how this can be installed as a project using pip
- Explain why it is good to be a cli tool, as it can be piped and used like any other command (wz, fzf, cp)
- How it is installed
- How the user can use it
- Unit testing and integration testing
- Documentation and version control strategies
- Explain the s-expressions extractor
- How they are loaded into python
- How they are stored for future use
- How the compilation uses graph dot

CHAPTER 7

Testing and Success Measurement

As well as documenting system testing, this section should also describe any unit testing or integration testing performed. If you are not familiar with unit, integration or system testing then it would be a good idea to investigate these notions and consider about how they relate to your project. This section might also detail any performance, reliability or usability testing performed, with quantification, i.e., numeric measurements, being used wherever possible. All those points are systems focused through. If you're doing something that's research focused or more of a social / analytical study then you need to think about how you'll measure success.

- Measure compilation time
- Measure time to build graph
- Measure time to make queries

CHAPTER 8

Project Management

What is the timeline for your project? What software development methodology will you use? Can you identify the major milestones? These types of questions are important for project planning and should be addressed directly.

- Gantt chart
- Weekly updates with supervisor
- Getting the s-expression extractor was a major milestone
- Using python instead of clojure was a significant improvement

CHAPTER 9

Evaluation

Usually you evaluate your project with regard to the functional and non-functional requirements you set out in the earlier chapter. This doesn't necessary mean that your project was successful but, if these requirements were appropriately specified, you it's likely that your project was successful. You might be reiterating some points from the Testing and Success Measurement chapter in your Project Report.

- Mention compilation time reduction, and improvements
- Mention how some queries take too long but were all implemented
- Mention how trees take some time but can be re-used instantaneously
- Creates graph from any agda project

CHAPTER 10

Conclusion

Generally speaking, section can take different forms - as a minimum you would normally provide a brief summary of your project work and a discussion of possible future work. You may also wish to reiterate the main outcomes of your project and give some idea of how you think the ideas dealt with by your project relate to real-world situations, etc.. For the Proposal, don't mention future work but do summarise your document.

- Compilation time could be further reduced
- A clever way of removing cycles could lead to faster queries
- Many useless nodes that are part of Agda's backend but not necessary for the user.
- Work on making the whole process more automatic and easily distributable

Bibliography

[1] what is s-expression?

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

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

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

[5] 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.

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

[16] 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.

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

CHAPTER 11

Appendix

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