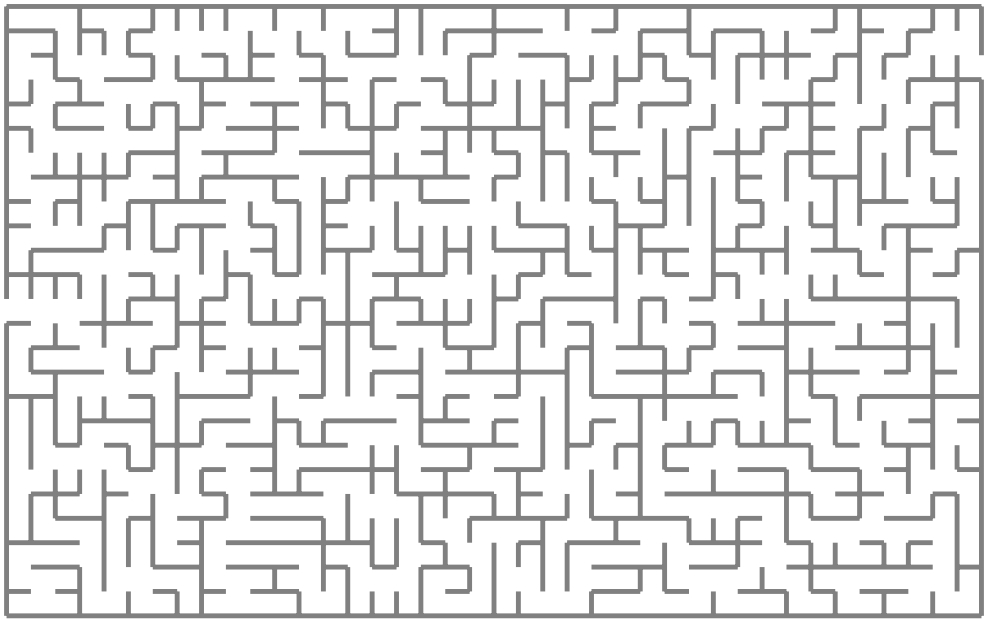
Can The Language Server Protocol Handle Dependent Types?

Master's Thesis



Willem Stuijt

Can The Language Server Protocol Handle Dependent Types?

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER & EMBEDDED
SYSTEMS ENGINEERING

by

Willem Stuijt
born in Santiago, Chile

Programming Languages Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Cover picture: Random maze.

Can The Language Server Protocol Handle Dependent Types?

Author: Willem Stuijt
Student id: 4995295

Abstract

The Language Server Protocol (LSP) is a protocol that standardizes the way Integrated Development Environments (IDEs) and text editors communicate with language servers to provide language-specific features like autocompletion, go-to-definition, and diagnostics. While LSP has been widely adopted by mainstream programming languages, its adoption in dependently typed languages has been slower due to the unique challenges posed by their complex type systems and interactive theorem proving capabilities. This thesis explores the potential of LSP for enhancing the development of dependently typed programs, focusing on the Agda programming language. We present the implementation of a prototype LSP server for Agda that leverages scope checking to provide fast and responsive IDE features. We evaluate the performance of the prototype and compare its feature completeness with existing Agda development tools. Our findings demonstrate that scope checking can serve as a foundation for implementing efficient LSP features in Agda, offering a promising direction for improving the tooling and overall development experience for dependently typed languages.

Thesis Committee:

Chair: Dr. J.G.H. Cockx, Faculty EEMCS, TU Delft
Committee Member: Ir. D.A.A. Pelsmaeker, Faculty EEMCS, TU Delft
Committee Member: Dr. Ing. S. Proksch, Faculty EEMCS, TU Delft

Preface

My journey into the fascinating world of programming languages began during my bachelor's degree when I took the "Concepts of Programming Languages" course taught by Casper Poulsen. This course sparked my curiosity and ignited a passion for exploring the intricacies of programming languages. It laid the foundation for my decision to delve deeper into this subject matter and ultimately led me to pursue this thesis.

Pursuing this passion, I embarked on the research process that ultimately led to this thesis. I would like to express my sincere gratitude to my advisors, Jesper Cockx and Daniël Pelsmaeker, for their invaluable guidance, support, and expertise throughout this journey. Their insights and feedback have been instrumental in shaping the direction and content of this work. I am also grateful to Sebastian Proksch for being a part of my thesis committee and providing valuable input.

Willem Stuijt
Delft, the Netherlands
June 20, 2024

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Agda LSP . . . 3
- 1.2 Research Objectives . . . 5

2 Language Server Protocol 7
- 2.1 Overview of Language Server Protocol . . . 7
- 2.2 Core Components of LSP . . . 7
- 2.3 Importance of LSP Features According to Developers . . . 10
- 2.4 Relevance to Dependently Typed Languages . . . 11
- 2.5 LSP in Dependently Typed Languages . . . 12
- 2.6 Contributions to Language Adoption . . . 12

3 Related Work 15
- 3.1 IDE Support for Dependently Typed Languages . . . 15
- 3.2 Challenges and Techniques for Responsive IDE Support . . . 16
- 3.3 Specification Language Server Protocol (SLSP) . . . 17
- 3.4 Conclusion . . . 17

4 Agda Implementation in Haskell 19
- 4.1 Parser and Concrete Syntax Tree (CST) . . . 19
- 4.2 Scope Checking and Abstract Syntax Tree (AST) . . . 21
- 4.3 Type Checking and Interaction . . . 22

5 Agda LSP Implementation 25
- 5.1 Scope Checking Approach . . . 25
- 5.2 Implementation Details . . . 29

6 Evaluation 35
- 6.1 Performance Evaluation . . . 35
- 6.2 Feature Completeness . . . 40

7 Discussion **45**
7.1 Scope Checking as a Foundation for LSP Features 45
7.2 Comparison with Existing Agda Development Tools 46
7.3 The Role of Emacs in Dependently Typed Language Development 47
7.4 Future Work and Improvements 47

8 Conclusion **49**

Bibliography **51**

List of Figures

1.1 IDE language integration before LSP 1
1.2 IDE language integration after LSP 2
1.3 Rename Refactoring 3
1.4 Rename Refactoring with shadowing 4
1.5 Implicit Argument Removal Refactoring 4
1.6 Implicit Argument Insertion Refactoring 5

4.1 High-level overview of Agda's front-end architecture 19

5.1 IPattern Data Type Structure 31
5.2 Implicit Argument Insertion Data Type 32
5.3 Implicit Argument Removal Data Type 33

6.2 Bar graph comparing the average execution times of each phase across all files in the Agda standard library. 37
6.3 Box plot showing the distribution of execution times for each phase on each file of the Agda standard library 38
6.5 Graph of line count against runtime for different phases of the Agda compiler . . 39

List of Tables

6.1 Line count and average execution times for each component on selected Agda files from the standard library. . . . 36
6.4 Line count and execution times for each component on the files of the Agda standard library that were slowest to scope check. . . . 38
6.6 Line count and execution times for each component on the files of the Agda standard library that were slowest to type-check. . . . 39
6.7 Editor Integration Feature Comparison . . . 40
6.8 Code Navigation Feature Comparison . . . 41
6.9 Code Assistance Feature Comparison . . . 41
6.10 Refactoring Feature Comparison . . . 42
6.11 Error Handling Feature Comparison . . . 42
6.12 Code Formatting Feature Comparison . . . 43
6.13 Agda-Specific Feature Comparison . . . 43

Chapter 1

Introduction

The adoption of programming languages is increasingly influenced by the quality of their tooling (Meyerovich and Rabkin 2013). What drives developer adoption is no longer solely based on a language's technical merits. Today, the quality of its tooling ecosystem plays a critical role in attracting and retaining users. This is especially true in the domain of interactive theorem proving languages like Coq, Lean, and Agda, where efficiency and user experience directly translate into productive research and development. One of the most important components of good tooling is the level of support provided by Integrated Development Environments (IDE). Good IDE integration provides the developer with all sorts of editor services, such as syntax highlighting, auto-completions, inline diagnostic messages and automated refactorings.

Historically, IDE and text editor integration for programming languages was a very time consuming process for language tooling developers. Each IDE and text editor would have it's own bespoke protocol, DSL or plugin system in which the integration would have to be written in. So a large subset of the programming language would have to be rewritten to support an IDE. And since each IDE had their own plugin system, a different implementation would be required for each IDE. This would be very time consuming but also very prone to errors, as each implementation could have subtle implementation differences. This is illustrated in Figure 1.1.

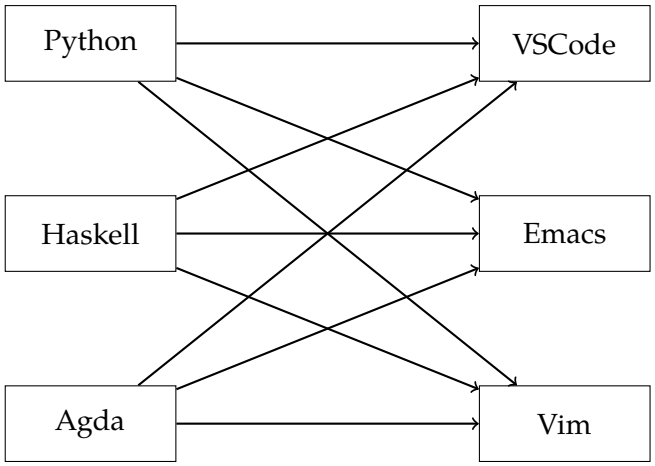


Figure 1.1: Old way of integrating languages with IDEs: Each language requires a separate integration for each IDE, resulting in a complex and error-prone process with lots of redundant work. The choice of languages and editors is just for illustrative purposes.

In 2016, Microsoft introduced the Language Server Protocol (LSP) to standardize the communication between code editors and language servers. A language server is a program that analyzes source code and provides language-specific features such as auto-completion,

go-to-definition, and error diagnostics. LSP established a common protocol based on JSON-RPC, allowing language servers to communicate with any compatible editor or IDE, regardless of its specific architecture or plugin system. By adopting LSP, editors would implement a client that understands the protocol, while language maintainers would only need to implement a single server that provides LSP-compliant information, as illustrated in Figure 1.2. This revolutionized the language integration process. Eliminating the need to rewrite language support for each individual editor saves development time and resources. Now virtually all moderately popular programming languages have at least one LSP implementation.

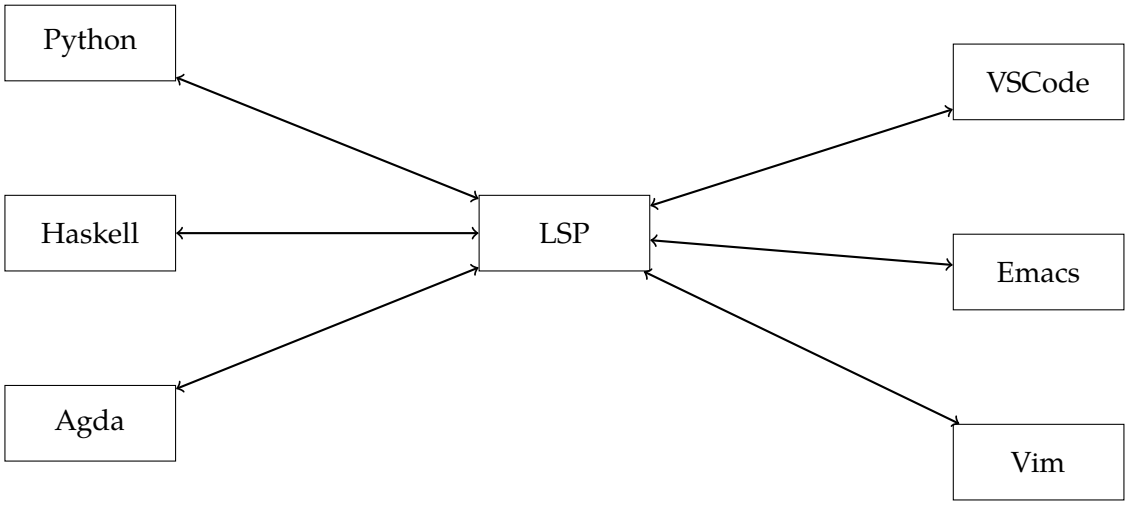


Figure 1.2: New way of integrating languages with IDEs using LSP: Each language offers an implementation of a language server which follows the LSP protocol. Each IDE or editor implements a client for the LSP protocol. This significantly reduces the number of direct integrations needed, simplifying the process and reducing errors.

But there is an exception to this, dependently typed languages are notoriously lacking behind in LSP adoption. Out of the most well known dependently typed languages Agda, Coq, Idris2 and Lean, only Idris2 (Idris Community 2024a) and Lean (LeanProver 2024) have LSP support, and they only support a small subset of the full LSP capabilities. This is likely due to the complex nature of these languages, which makes implementing a comprehensive language server a challenging task. Dependently typed languages often have intricate type systems and advanced features that require sophisticated analysis, this makes them slower at type-checking than typical programming languages. They also support advanced interactions with the type-checker that are not found in mainstream languages, making it challenging to provide efficient and interactive IDE support.

One example of such a dependently typed language is Agda, which has gained popularity in the research community due to its powerful type system and interactive development environment. Being dependently typed means that Agda allows types to depend on values, enabling the expression of more precise properties and invariants within the type system itself. This allows, for example, expressing mathematical proofs in the language. In this thesis, I focus on Agda as a representative example of a dependently typed language, but the insights gained can be applied to other languages in this domain as well.

At the time of writing, when someone opens an Agda file in most IDEs or text editors, they are met with monocolored text, no completions, no diagnostics while typing and no interactivity. This is a stark difference from what a typical experience would be for mainstream languages. Agda has no LSP implementation, but Agda does have a more unique approach to editor integration. Agda's most well supported editor is Emacs, where a dedicated mode called "agda-mode" (Coquand, Takeyama, and Synek 2006) offers some features like those found in LSP-backed editors. For example, agda-mode provides advanced syntax highlight-

ing that makes use of typing information. However, the program needs to be type-checked first to benefit from this feature, and the user must prompt it, it does not happen automatically. Most importantly it provides a unique command-based interaction loop where users interact directly with the type-checker by automatically placing and filling "holes" in the active source code. This is a unique feature for languages with powerful typesystems like Agda. Some examples of this workflow:

- **Type inference:** Agda allows users to leave placeholders, known as "holes", in their code to represent incomplete or unknown expressions. These holes can be used for both type signatures and implementations. By leaving a hole, users can continue writing code without being blocked by incomplete details. Agda-mode interacts with the type-checker to provide feedback on the expected type of each hole based on its context. This guides users towards correct type annotations and helps them complete the implementation incrementally.
- **Interactive Program and Proof Editing:** In Agda, programs and proofs are essentially the same, both being constructions that satisfy types. Agda-mode leverages this by allowing users to incrementally build both programs and proofs, suggesting possible next steps based on type information and context. This facilitates an exploratory process where developers can gradually refine their solutions while receiving immediate feedback on the validity of each step.
- **Goal-Directed Development:** Whether you're building a program or proving a theorem, Agda-mode supports a goal-oriented workflow. Instead of writing everything at once, users can decompose a complex task into smaller, manageable goals (or subgoals). Agda-mode aids in navigating and tracking these subgoals, making it easier to focus on individual parts of the problem while maintaining an understanding of the overall structure.

1.1 Agda LSP

As part of this research, we implemented an experimental Language Server Protocol (LSP) server for Agda. The LSP integration for Agda offers several features that significantly improve the development experience for Agda programmers. These features include variable renaming, diagnostic and removal of unused implicit arguments, and insertion of explicitly used implicit arguments.

Variable renaming

Variable renaming is a crucial feature that allows programmers to change the name of a variable throughout the codebase consistently. Consider the example in Figure 1.3

```
_*_ : Nat → Nat → Nat
zero    * n  =  zero
(suc m) * n  =  n + (m * n)
```

(a) Before Refactoring

```
_*_ : Nat → Nat → Nat
zero    * n  =  zero
(suc m) * x  =  x + (m * x)
```

(b) After Refactoring

Figure 1.3: Code before and after automated renaming of the `n` variable to `x`

Suppose we want to rename the variable `n` in the second clause to `x`. Manually performing this transformation can be tedious and error-prone, especially when the function is large and the variable is used in multiple places. The LSP server for Agda simplifies this process by

providing a rename refactoring feature. With a single action, the variable can be renamed consistently throughout the code.

Implementing variable renaming can be challenging due to the need to handle scoping rules, name shadowing, and ensuring the correctness of the transformation across the entire codebase. Take another example where one of the variables is shadowed, as shown in Figure 1.4.

Here, the variable `outer` in the function `shadowExample` shadows the top-level declaration of `outer`. When renaming occurs, only the names referring to the internal variable should be renamed.

```
outer : Nat
outer = 3

shadowExample : Nat → Nat
shadowExample outer = outer + 2
```

(a) Before Refactoring

```
outer : Nat
outer = 3

shadowExample : Nat → Nat
shadowExample x = x + 2
```

(b) After Refactoring

Figure 1.4: Code before and after automated renaming of `outer` variable to `x` without affecting the shadowed `outer` declaration.

Diagnostic and removal of unused implicit arguments

Agda allows the use of implicit arguments, which are inferred by the compiler and do not need to be explicitly provided by the programmer, although they can be made explicit. However, sometimes implicit arguments may become unused, leading to unnecessary clutter in the code. Consider the example in Figure 1.5.

```
id : {A : Set} → A → A
id {A} x = x
```

(a) Before Refactoring

```
id : {A : Set} → A → A
id x = x
```

(b) After Refactoring

Figure 1.5: Code before and after automated removal of the unused implicit argument `A`.

In this case, the implicit argument `A` is unused. The LSP server for Agda detects such unused implicit arguments and provides a diagnostic message, underlining the declaration with a yellow line and displaying the message "Unused implicit variable". Moreover, it offers a quick fix that automatically removes the unused variable declaration.

Detecting and removing unused implicit arguments helps keep the code clean and maintainable. However, it requires careful analysis of the usage of variables throughout the program. The LSP server for Agda handles this complexity by utilizing scope information to know which implicit parameters are available in the given context and which ones are not being used.

Insertion of explicitly used implicit arguments

In some cases, implicit arguments need to be used explicitly within the function body. Consider the example in Figure 1.6.

```
appTwice : {A : Set} → {f : A → A} → A → A
appTwice x = f (f x)
```

(a) Before Refactoring

```
appTwice : {A : Set} → {f : A → A} → A → A
appTwice {f = f} x = f (f x)
```

(b) After Refactoring

Figure 1.6: Code before and after automated insertion of the explicitly used implicit argument `f`.

Here, we want to use the implicit parameter `f` explicitly to call it twice. However, `f` is not declared in scope, so we must make it explicit. The LSP server for Agda detects this situation and offers a quick fix to make `f` explicit.

Inserting explicitly used implicit arguments can be tricky, as it requires knowing which implicit variables are available within the context and which ones are already defined explicitly. Having the LSP server for Agda handle this automatically can smoothen the development process.

1.2 Research Objectives

While agda-mode provides great interactivity, it still lacks basic functionality that would be provided by an LSP compliant extension, such as inline auto-complete, hover information and automatic renaming. However, this unique workflow raises the question: can the strengths of agda-mode style workflows be combined with the wider reach and standardized approach of LSP? This research aims to explore the potential of LSP for enhancing the interactive development of dependently typed programs in general. This research will delve deeper using Agda as a specific example, but the insights gained will be applicable to other dependently typed languages due to the shared characteristics and challenges within this domain.

Developing a prototype LSP server for Agda serves as a valuable approach to better understand the challenges and opportunities associated with integrating LSP support for dependently typed languages. By implementing key LSP features and exploring their feasibility within the context of Agda, we can gain practical insights into the specific requirements, limitations, and potential solutions. This hands-on experience will provide a foundation for answering the main research questions and identifying strategies to effectively leverage LSP for enhancing the development experience of dependently typed programs. The insights gained from the prototype implementation will directly contribute to the field of language server protocol integration for dependently typed languages.

Contributions

- We identify the key LSP features that are beneficial for programming in dependently typed languages (Chapter 2).
- We investigate how other dependently typed languages integrate with LSP or their IDEs, and what lessons can be learned from their approaches (Chapter 3).
- We explore how existing language implementation libraries can be utilized to build LSP servers for dependently typed languages (Chapter 4).
- We identify the specific challenges and limitations encountered when implementing LSP features for dependently typed languages (Chapter 5).
- We discuss strategies for balancing performance considerations with the need for comprehensive and accurate LSP features (Chapter 5).

- We propose potential strategies for overcoming these limitations and achieving efficient LSP server implementations (Chapters 5 and 6).
- We implemented a prototype LSP server[1] for Agda which includes features such as auto-completions, find/go-to references, renaming and implicit parameter refactorings.

[1] https://github.com/willemstuijt/agda-lsp

Chapter 2

Language Server Protocol

2.1 Overview of Language Server Protocol

The Language Server Protocol (LSP) is a standardized protocol that defines how programming language features can be implemented and exposed to development tools, such as code editors and Integrated Development Environments (IDEs). The primary purpose of LSP is to provide a common interface between language-specific tools and various development environments, enabling developers to access rich language features regardless of their preferred editor or IDE.

LSP was first introduced by Microsoft in 2016 as part of their efforts to improve the development experience in Visual Studio Code (Microsoft 2023). The protocol was designed to be language-agnostic and editor-agnostic, allowing for a more modular and flexible approach to language tooling. Since its inception, LSP has gained widespread adoption across the software development community, with numerous programming languages and development tools implementing support for the protocol, namely: Visual Studio Code, JetBrains IDEs such as IntelliJ and Fleet (JetBrains 2024), Vim (Shrestha 2024), and many others.

The development of LSP has been driven by the need to address the challenges associated with traditional language tooling approaches. Prior to LSP, language features were often tightly coupled with specific editors or IDEs, requiring developers to use a particular tool to access advanced functionality. To support more editors or IDEs it would be often necessary to rewrite a significant part of the frontend of a language. This fragmentation of language tooling led to inconsistent user experiences and limited the ability of developers to work with their preferred tools.

LSP aims to solve these problems by decoupling language features from the development environment. As discussed in Section 2.2, LSP provides a standardized interface for communication between the language server and the client (editor or IDE). This enables language tooling to be developed independently of the client, allowing for greater flexibility and interoperability. As a result, language servers can be reused across multiple development environments, and clients can support multiple languages without the need for language-specific integrations.

2.2 Core Components of LSP

The Language Server Protocol follows a client-server architecture, where the client is typically a code editor or an Integrated Development Environment (IDE), and the server is a language-specific tool that provides language features. This separation of concerns allows for a modular and extensible design, enabling language tooling to be developed independently of the client.

In the LSP architecture, the client communicates with the server using JSON-RPC, a lightweight remote procedure call protocol that uses JSON as its data format. The client sends requests to the server for various language features, such as code completion, hover information, or code actions. The server processes these requests and sends back responses containing the requested information or actions.

The communication between the client and the server is bidirectional, meaning that the server can also send notifications to the client to provide updates on the state of the language tooling. This could involve sending diagnostic messages to flag errors or warnings in the code as soon as the semantic analysis phases are done, or providing real-time updates on completion results as they become available, especially during lengthy calculations.

The Language Server Protocol defines a set of core features that language servers can implement to provide rich language functionality to clients. These features include:

Navigation

Go to Definition and Find References are essential features that greatly enhance code navigation capabilities. Go to Definition enables developers to jump directly to the definition of a symbol, such as a function, class, or variable, by simply clicking on the symbol or using a keyboard shortcut. This feature saves time and effort in navigating through large codebases and understanding the implementation details of specific code elements. Find References provides a comprehensive list of all the locations where a particular symbol is referenced within the project. This feature is invaluable for understanding the usage and impact of a symbol, as well as for performing code analysis tasks.

Renaming

Rename refactoring is a powerful feature that allows developers to safely rename symbols across the entire project while ensuring that all references to the symbol are updated accordingly. This feature helps maintain code consistency and prevents potential errors that may arise from manual renaming.

Diagnostic Messages

Diagnostic messages in LSP are called Diagnostics. They play a crucial role in providing developers with real-time feedback about potential issues in their code. Two key diagnostic features are warnings for unused variables and fast error feedback.

Unused variable warnings help developers identify and eliminate unnecessary declarations, improving code clarity and maintainability. By highlighting variables that are declared but never used, the language server can guide developers towards writing cleaner and more efficient code.

Fast error feedback is essential for a smooth development experience, enabling developers to catch and fix errors early in the coding process. The language server can analyze the code in real-time and provide immediate feedback on syntax errors, type mismatches, and other common programming mistakes. This feature helps developers identify and resolve issues quickly, reducing the time spent on debugging and improving overall productivity.

Completions

Code completion is a vital feature for enhancing developer productivity and facilitating codebase exploration. It provides developers with suggestions for available functions, types, and modules based on the current context, helping them discover and use relevant identifiers quickly.

Language servers can implement intelligent code completion by leveraging the semantic understanding of the codebase. They can analyze the context in which the developer is typing and provide accurate and context-aware suggestions. This feature not only saves typing effort but also helps developers explore the available APIs and discover new functionality.

Code completion can be further enhanced with additional information, such as parameter hints, return types, and documentation snippets, to provide a more comprehensive and informative coding experience.

Code Lens

Code Lens is a powerful feature that provides contextual information and actionable insights directly within the editor. It can display relevant information and actions directly above or below the corresponding code elements, offering a more intuitive and streamlined workflow for developers. Some examples of the information that can be displayed in a Code Lens include:

- Code Lens can show the type information for variables, parameters, and return values, making it easier to understand the expected types and catch potential type-related issues.
- Code Lens can indicate the status of associated unit tests, showing whether the tests are passing or failing, and providing quick access to run or debug the tests.
- Code Lens can display code metrics, such as cyclomatic complexity (a measure of the number of linearly independent paths through a program's source code) or lines of code, helping developers assess the complexity and maintainability of their codebase (McCabe 1976).

Code Actions

Code Actions offer automated code transformations and quick fixes for common programming tasks. They can assist with tasks such as handling implicit arguments, suggesting code improvements, and applying language-specific refactorings.

When the language server detects a particular code pattern or issue, it can provide a set of Code Actions to the client. These actions appear as clickable suggestions or light bulbs in the editor, allowing developers to apply the suggested changes with a single click.

Code Actions can range from simple transformations, such as adding missing imports or removing unused variables, to more complex refactorings, like extracting a code block into a separate function or converting between different coding styles.

Semantic Tokens

Editors like VSCode and Emacs rely on regular expressions or textmate grammars (MacroMates Ltd. 2024) for syntax highlighting. These approaches work by matching patterns in the code and applying colors based on predefined rules. While this method is sufficient for simple languages, it falls short when dealing with more complex languages or advanced coding constructs.

The limitations of regex-based syntax highlighting become evident in scenarios where the meaning of a token depends on its context, scoping or type information. For example, in many languages, a word might be a keyword in one context but a variable or function name in another. Regular expressions alone cannot accurately distinguish between these cases, leading to inconsistent or incorrect syntax highlighting.

This is where Semantic Tokens come into play. Semantic Tokens offer detailed semantic information for syntax highlighting, enabling more accurate and context-aware coloring

of code elements. This feature provides an alternative to regular expressions and textmate grammars for syntax highlighting, which may be challenging to implement for languages with complex syntactic structures.

With Semantic Tokens, the language server can provide fine-grained semantic information about each token in the code, such as its type (e.g., keyword, variable, function), scope (e.g., local, global), and modifiers (e.g., static, readonly). The client can then use this information to apply more sophisticated and meaningful syntax highlighting, improving code readability and comprehension.

Other LSP Features

- **Hover:** This functionality displays contextual information about a symbol when it is hovered over with the cursor, including documentation, type information, and other relevant details. Hover information helps developers understand the purpose and usage of code elements without having to navigate away from their current context.

- **Type Hierarchy:** This feature visualizes the relationships between types, displaying both super types (parent types) and sub types (child types). Type Hierarchy helps developers understand the inheritance structure of classes and interfaces, making it easier to navigate and comprehend complex type hierarchies.

- **Signature Help:** This feature displays information about function signatures and parameters, including their names and types, as function calls are typed. Signature Help assists developers in understanding the expected arguments and their types, reducing the chances of passing incorrect arguments or missing required parameters.

- **Document Links:** This feature detects and highlights links within a document, such as URLs or file paths. Document Links enable developers to easily identify and navigate to referenced resources, improving the discoverability and accessibility of related information.

- **Document Highlighting:** This functionality highlights all occurrences of a selected symbol within the current file. Document Highlighting helps developers quickly identify and locate all instances of a particular code element, making it easier to understand its usage and impact throughout the file.

These features form the core of the Language Server Protocol and provide a foundation for building rich language tooling experiences. Language servers can implement additional features beyond this core set, depending on the specific needs and characteristics of the language they support.

The modular nature of LSP allows for a gradual and incremental approach to implementing language features. Language server developers can prioritize and implement the features that are most relevant and impactful for their language, while also having the flexibility to extend and customize the functionality as needed.

2.3 Importance of LSP Features According to Developers

Several studies have investigated the importance of various IDE features to developers. These studies provide valuable insights into which features are most valued and how they contribute to the overall development experience.

Amann et al. (2016) found that code completion was the most frequently used assistance tool in Microsoft's Visual Studio IDE, followed by the build system, debugger, and navigation

tools (Amann et al. 2016). This highlights the importance of features like code completion, diagnostics, and code navigation in enhancing developer productivity and code quality.

Zayour and Hajjdiab (2013) emphasized the impact of modern IDE features such as Intellisense and code navigation on reducing accidental difficulties, particularly syntax errors and code navigation challenges (Zayour and Hajjdiab 2013). This underscores the value of LSP features that support code completion, error checking, and navigation in streamlining the development process and improving overall efficiency.

Murphy et al. (2006) revealed that developers heavily rely on editors and specific views within the Eclipse IDE, such as the Package Explorer and Console (Murphy, Kersten, and Findlater 2006). They also observed frequent use of keyboard shortcuts for executing commands and navigating through code. This highlights the need for LSP features that support efficient code editing, navigation, and interaction with the development environment.

The studies mentioned above focused primarily on imperative and object-oriented programming languages. While their findings are still relevant to functional and dependently typed languages like Agda, some differences in developer preferences and usage patterns should be acknowledged. For example, the significance of type hierarchy exploration, a feature commonly used in object-oriented languages, may be less pronounced in Agda due to its different type system. Conversely, the importance of features like code actions, which can assist with implicit arguments and type-related refactorings, may be heightened in Agda.

Understanding the importance of these LSP features in the context of dependently typed languages can help in prioritizing the development of language tooling and ensuring that the most valuable functionalities are implemented first.

2.4 Relevance to Dependently Typed Languages

The findings from these studies suggest that the most important LSP features for developers are code completion, diagnostics, hover information, and code navigation. These features are particularly relevant for dependently typed languages, which often have complex type systems and require a deep understanding of the relationships between types and terms.

Code completion is crucial for dependently typed languages, as it can help developers navigate the complex type landscape and discover available functions and types based on the current context. By providing context-aware suggestions, code completion can reduce the cognitive burden of remembering and manually typing out complex type signatures.

Diagnostics are also essential for dependently typed languages, as they can help catch type errors early in the development process. Given the intricate nature of dependent types, even small type mismatches can lead to significant issues down the line. By providing real-time feedback on type errors and other issues, diagnostics can help developers maintain the correctness and consistency of their code.

Hover information is particularly useful for dependently typed languages, as it can provide valuable insights into the types and relationships of symbols in the code. By displaying type information and documentation on hover, developers can quickly understand the role and purpose of various components without having to navigate away from their current context.

Code navigation features, such as go to definition and find references, are important for dependently typed languages, as they enable developers to explore the complex relationships between types and terms. By providing fast and accurate navigation to the definitions and references of symbols, these features can help developers understand the structure and flow of their code, making it easier to reason about and modify.

Code Lens and Code Actions offer additional benefits specific to dependently typed languages. Code Lens can provide contextual information and actionable insights directly within the editor, potentially replacing the need for command-based interaction with holes in Agda.

It can display inferred types, suggest possible solutions, and offer quick actions to fill holes, streamlining the development workflow. Code Actions, on the other hand, can automate code transformations and provide quick fixes for language-specific constructs, such as handling implicit arguments in Agda.

Furthermore, the Semantic Tokens feature is particularly suitable for dependently typed languages like Agda, which often have complex syntactic structures. The alternative to this feature for syntax highlighting is using tree-sitter based grammars. But Agda's grammar is difficult to express using tree-sitter due to its use of complex syntactic structures such as mixfix operators (Danielsson and Norell 2011). This makes the semantic tokens feature of LSP ideal as it allows syntax highlighting to be offered through the use of Agda's official parser implementation.

2.5 LSP in Dependently Typed Languages

Dependently typed languages, such as Agda, Coq, and Idris, pose unique challenges for Language Server Protocol implementations due to their complex type systems and proof-oriented nature. These languages often require more advanced type checking and theorem proving capabilities compared to traditional programming languages.

One of the primary challenges in implementing LSP for dependently typed languages is the need for fast type checking. Dependently typed languages rely heavily on advanced type inference and unification, which can be computationally expensive, especially for large codebases. To provide a responsive and interactive development experience, language servers for dependently typed languages must be able to perform type checking and inference in real-time, without introducing significant latency or performance overhead.

Another challenge is the integration of proof assistance and interactive theorem proving into the LSP workflow. Dependently typed languages often include features such as proof tactics, hole-driven development, and interactive proof refinement. These features require a tight integration between the language server and the proof engine, as well as the ability to provide real-time feedback and guidance to the user during the proof development process.

Additionally, dependently typed languages often have a rich and expressive syntax, with features such as dependent pattern matching and implicit arguments. Language servers for these languages must be able to accurately parse and analyze this syntax, while also providing helpful error messages and suggestions for resolving type errors and proof obligations, and even suggesting quick fixes or refactorings for common patterns in theorem proving code.

2.6 Contributions to Language Adoption

The adoption of the Language Server Protocol for dependently typed languages can also contribute to the broader adoption and usability of these languages in industry and academia.

Historically, dependently typed languages have been perceived as complex and challenging to learn and use (Ringer et al. 2019), due in part to the lack of mature tooling and editor support, or at least for the stark difference between dependently typed workflows and mainstream programming language workflows. By providing a more standard and interactive development experience through LSP, these languages can become more accessible and attractive to a wider range of developers, from undergraduates to those working on real-world applications and research projects.

The improved tooling and editor support enabled by LSP can also help to showcase the unique benefits and capabilities of dependently typed languages, such as the ability to express complex type constraints and invariants, and to guide the user in proving properties of programs. By making these features more discoverable and easier to use, LSP can help to

demonstrate the value and potential of dependently typed programming to a broader audience.

Overall, the integration of Language Server Protocol into development environments for dependently typed languages can play a significant role in promoting the adoption, usability, and impact of these languages. By providing a more accessible, efficient, and interactive development experience, LSP can help to unlock the full potential of dependently typed programming and enable more developers to leverage the power and expressiveness of these languages in their work.

Chapter 3

Related Work

The field of dependently typed programming languages has seen significant advancements in recent years, with a growing focus on improving the development experience through better IDE support. This thesis builds upon prior research on language servers, IDEs for dependently typed programming languages, and techniques for enhancing the development experience with advanced type systems. In this chapter, we will explore the most relevant work in each of these areas, highlighting the challenges and opportunities that arise when integrating dependently typed languages with modern development tools.

The Language Server Protocol (LSP) has emerged as a standardized way to provide language-specific features in IDEs, enabling developers to work with their preferred tools while benefiting from rich language support. However, LSP support for dependently typed languages has been limited to date. This is partly due to the complex nature of these languages, which rely on advanced type systems and proof-based reasoning, making it challenging to provide the real-time feedback and interactive features expected in modern IDEs.

3.1 IDE Support for Dependently Typed Languages

3.1.1 Agda

Despite the limited LSP support, there have been notable efforts to improve the IDE experience for dependently typed languages. Agda, a dependently typed functional programming language, has a well-established Emacs mode (Coquand, Takeyama, and Synek 2006) that provides a command-based interaction model for querying the type checker and incrementally developing proofs. While this Emacs mode offers a solid foundation for working with Agda, it lacks many features expected in modern IDEs, such as auto-completion and real-time diagnostics.

3.1.2 Coq

The Coq ecosystem has seen significant advancements in IDE support. The company-coq (C. F. Pit-Claudel, Courtieu, and C. Pit-Claudel 2016) package provides an IDE-like experience within Emacs, offering features such as auto-completion, real-time documentation, and interactive proof development. This demonstrates the potential for integrating advanced language features with traditional IDE functionalities, enhancing the overall development experience. However, company-coq is still limited to the Emacs environment, which may not be the preferred choice for all developers.

In addition to company-coq, there are other alternative environments for Coq development. CoqIDE (*CoqIDE* n.d.) is a standalone IDE provided by the Coq team, which offers a simplified interface for writing and evaluating Coq code. It includes features such as syntax highlighting, error reporting, and interactive proof development. Another option is Proof

General (*Proof General* n.d.), a generic interface for proof assistants that supports Coq. It provides an Emacs-based environment with similar features to CoqIDE and company-coq.

The Coqoon (Faithfull et al. 2018) project has taken a different approach by integrating Coq with the Eclipse IDE. Coqoon provides project management, syntax highlighting, and interactive proof stepping, bringing Coq development closer to the mainstream IDE experience. However, it is important to note that Coqoon is a separate IDE rather than a general-purpose solution that brings Coq support to existing IDEs. This is due to the challenge of integrating dependently typed languages with established development environments, as it often requires significant effort to adapt the language's unique features to the IDE's architecture.

3.1.3 Lean

A promising example of LSP's potential for dependently typed languages can be found in the Lean 4 theorem prover. The Lean LSP server implementation (Moura and Ullrich 2021) powers the Lean VS Code extension, providing a rich IDE experience that includes syntax highlighting, type information, auto-completion, and jump-to-definition. This demonstrates that LSP is well-suited for handling the rich semantic information used by dependently typed languages, opening up new possibilities for integrating these languages with a wide range of IDEs.

Building on the success of the Lean LSP server, there has been further work on improving the user experience for mathematicians working with the Lean language. Nawrocki et al. (Nawrocki, Ayers, and Ebner 2023) have developed a more friendly user interface that aims to make Lean more accessible to mathematicians who may not have extensive programming experience. This highlights the importance of considering the target audience when designing IDE support for dependently typed languages, as the needs of mathematicians and programmers may differ significantly.

3.1.4 Idris and Idris 2

Idris has also made strides in improving its IDE support. The Idris mode for Emacs (Mehnert and Christiansen 2014) provides a command-based interaction model similar to that of Agda, allowing users to query the type checker and incrementally develop proofs. Despite the lack of LSP support in Idris 1, the community has developed various IDE integrations to enhance the development experience. For example, there are plugins available for popular IDEs such as Atom and Visual Studio Code.

Idris 2 (Brady 2021), the successor to Idris, has been developed with a focus on improving the language's performance and expressiveness. Idris 2 also aims to enhance the developer experience by providing better tooling and documentation. One of the notable improvements in Idris 2 is the introduction of an LSP server (Idris Community 2024b), which enables a more comprehensive and robust IDE integration. One of the more interesting features of the Idris 2 LSP is its use of Code Actions to provide interactive editing capabilities, such as case splitting, refining holes, and generating function clauses based on type signatures.

3.2 Challenges and Techniques for Responsive IDE Support

One of the major challenges when implementing an LSP-compliant server for a dependently typed language is the slow type-checking times. Even when imported modules' types are cached, checking a single file in Agda can take seconds, which is unacceptable for the real-time feedback expected in modern IDEs. To address this issue, researchers have explored techniques for improving type-checking time, such as incremental parsing (Collins and Roark 2004; Wagner and Graham 1998) and incremental type checking (Zwaan, Antwerpen, and Visser 2022; Pacak, Erdweg, and Szabó 2020). These techniques allow for faster feedback by

reusing previously computed syntax and type information within the same file, reducing the overhead of re-checking unchanged code.

Incremental type checking has been successfully employed in the Lean LSP server (Moura and Ullrich 2021) to provide responsive type checking as the user edits their code. By caching and reusing type information, the Lean LSP server is able to provide real-time feedback even for complex proofs. This demonstrates the feasibility of applying incremental techniques to dependently typed languages, paving the way for more responsive IDE support.

3.3 Specification Language Server Protocol (SLSP)

Another relevant area of research is the Specification Language Server Protocol (SLSP) (Rask et al. 2021), which aims to extend LSP to support specification languages. The authors argue that while LSP has been successful for programming languages, it lacks support for features specific to specification languages, such as proof obligation generation and theorem proving. The SLSP aims to fill this gap by providing standardized extensions for these features, promoting decoupling of language support from IDEs, and reducing the effort required to integrate specification languages into different IDEs.

In the context of formal methods, proof obligations are logical statements that need to be proven to ensure the correctness of a specification. The SLSP defines a "generate" message that allows the client (IDE) to request proof obligations from the server (SLSP). The server responds with a list of proof obligations, each containing an ID, name, type, location, and an optional flag indicating if it has been proven. A notification message is also defined to keep the proof obligations synchronized with the specification, ensuring that the client can request updated proof obligations if the specification changes.

Theorem proving is the process of formally verifying the correctness of mathematical statements or logical formulas. The SLSP includes messages to support theorem proving features. The protocol defines messages for:

- querying the available lemmas (formally proven statements) in a specification;
- initiating a proof session for a lemma;
- applying automated theorem proving to try and prove a lemma automatically;
- sending commands for interactive theorem proving, where a user guides the proof process;
- undoing proof steps;
- getting a list of available prover commands.

The SLSP aims to provide a standardized way for IDEs to interact with different theorem provers, abstracting away the specific commands and protocols of each prover.

3.4 Conclusion

While LSP support for these languages has been limited to date, there have been notable efforts to improve the development experience through projects like agda-mode, company-coq package, Coqoon, and the Lean LSP server. The success of these projects demonstrates the potential for integrating dependently typed languages with modern IDEs, providing developers with the rich language support they need to work effectively.

The main challenge of providing responsive feedback in the face of slow type checking times still remains. Incremental parsing and type checking techniques have shown promise

in addressing this issue, as demonstrated by the Lean LSP server. The main problem with these advanced techniques is that they are difficult to implement and would require significant modifications to a dependently typed language's compiler to incorporate.

Chapter 4

Agda Implementation in Haskell

This chapter provides a brief overview of the Agda implementation in Haskell. The information here will be important to understand the implementation of the prototype LSP. Since the implementation mostly relies on the parser, Concrete Syntax Tree (CST), Abstract Syntax Tree (AST) and scope checking, details about type-checking will be omitted. First a general overview of the Agda implementation will be given. Then the parser and CST will be described in more detail. Then the AST and how it relates to scope checking. As well as an overview of the APIs Agda exposes for interacting with it. Figure 4.1 presents a high-level overview of Agda's front-end architecture, illustrating the main components and their interactions.

Figure 4.1: High-level overview of Agda's front-end architecture

4.1 Parser and Concrete Syntax Tree (CST)

The parser is the entry point of Agda's analysis pipeline, responsible for transforming the textual representation of Agda source code into a structured form known as the CST. The CST captures the syntactic structure of an Agda program, representing elements such as expressions (`Expr`), declarations (`Declaration`), patterns (`Pattern`), and more. It closely resembles the original source code, preserving the concrete syntax and layout of the program.

In essence, the parser transforms Agda code into a structured format, the CST, which serves as the foundation for subsequent scope analysis and interpretation of the code. This transformation is crucial for implementing LSP features like code completion, refactoring, and error diagnostics.

4.1.1 Expressions

The `Expr` datatype represents various forms of expressions in the CST. Some relevant constructors that are frequently used in the LSP implementation are `Ident` and `RawApp`.

Ident

The `Ident` constructor represents an identifier in an Agda expression. It takes a `QName` (qualified name) as its argument. A `QName` consists of either a single name or a module name and another nested `QName`, allowing for unambiguous references to identifiers defined in different modules. For example, `MyModule.MySubmodule.myFunction` is a qualified name that refers to the `myFunction` identifier defined in the `MySubmodule` submodule of the `MyModule` module.

More precisely, it consists of three nested `QNames`: `MyModule`, `MySubmodule`, and `myFunction`, where each subsequent `QName` is nested within the previous one. This hierarchical structure allows for unique identification of names within a module system. This expression is particularly important for code completion, as it enables precise suggestions based on the module context.

RawApp

The `RawApp` constructor represents a raw application in an Agda expression. It consists of a `Range` (source code location) and a non-empty list of expressions (`List2 Expr`). Unlike the fully parsed AST, the `RawApp` constructor does not have operators parsed yet. It simply captures the application of expressions without further analysis of the operator precedence and associativity.

Listing 4.1: Example Agda code with a raw application

```
f : Nat -> Nat -> Nat
f x y = x + y * 2
```

Consider the Agda code in Listing 4.1. In the CST, the expression `x + y * 2` would be represented as a `RawApp` node, containing the expressions `x`, `+`, `y`, `*`, and `2` as its arguments, without any further parsing of the operators.

4.1.2 Declarations

The `Declaration` datatype represents various forms of declarations in the CST. Some relevant constructors that are frequently used in the LSP implementation are `Import`, `Module`, `TypeSig` and `FunClause`.

Import

The `Import` constructor represents an import declaration in an Agda module. It consists of a `Range` (source code location), a `QName` (the module being imported), an optional `AsName` (for renaming imports), an `OpenShortHand` flag (indicating if the import is open or not), and an `ImportDirective` (specifying how the import should be handled).

Listing 4.2: Example Agda code with import declarations

```
module Main where

import Data.List as L
open import Data.Maybe
```

In the CST of the Listing 4.2, the first import declaration would be represented as an `Import` node with the `QName` `Data.List`, the `AsName` `L`, and the `OpenShortHand` flag set to `false`. The second import declaration would be represented as an `Import` node with the `QName` `Data.Maybe`, no `AsName`, and the `OpenShortHand` flag set to `true`.

Module

The `Module` constructor represents a module declaration in an Agda file. It consists of a `Range` (source code location), an `Erased` flag (indicating if the module has a run-time component or consists of compile-time proofs only), a `QName` (the name of the module), a `Telescope` (the module parameters), and a list of `Declarations` (the content of the module).

Listing 4.3: Example Agda code with a parameterized module

```
module Parameterized (A : Set) where

  data List : Set where
    nil  : List
    cons : A -> List -> List
```

The CST of the code in Listing 4.3 would be represented as a `Module` node with the `QName` `Parameterized`, the `Telescope` containing the parameter `A : Set`, and the list of `Declarations` containing the `List` datatype declaration.

TypeSig and FunClause

In the CST, type signatures (`TypeSig`) and function clauses (`FunClause`) are represented as separate declarations. This means that the function clauses for a given function are not yet grouped together with their corresponding type signature.

For example, consider the following Agda code:

Listing 4.4: Example Agda code with a type signature and function clauses

```
append : {A : Set} -> List A -> List A -> List A
append nil         ys = ys
append (cons x xs) ys = cons x (append xs ys)
```

The CST of the code in Listing 4.4 would be represented as three separate `Declaration` nodes: one `TypeSig` node for the type signature, and two `FunClause` nodes for the function clauses. The grouping of the type signature and function clauses happens later in the parsing pipeline.

4.1.3 Parser API

To call Agda's parser from Haskell one can use the `parseFile` function from the `Agda.Syntax.Parser`[1] module. You can pass various different parser configurations to this function depending on what you want to parse. For example to parse a single expression you can use `exprParser`. To parse the whole file the `moduleParser` can be used, this results in a parser that produces a CST Module which contains a list of pragmas and declarations which can be used for further analysis. To run a parser and get its result you can use `runPMIO`. This function takes in the constructed parser and runs it to completion, returning the expected result or a `ParseError`.

One of the significant limitations of Agda's parser is that it does not handle parsing errors gracefully. When Agda's parser encounters an error while parsing, it immediately gives up. It does not try to perform some error recovery and return a partially correct CST and a list of errors messages. This means that if there is a single syntax error, then no analysis can be done. This is a significant limitation to implementing LSP features as most of the time while users are editing files, they will be in syntactically invalid states, and in all these states most LSP features will stop working. Fixing this would allow for a much smoother LSP experience.

4.2 Scope Checking and Abstract Syntax Tree (AST)

Scope checking is a crucial phase in Agda's compilation process, serving as the bridge between the Concrete Syntax Tree (CST) and the Abstract Syntax Tree (AST). Its primary pur-

[1] `https://github.com/agda/agda/blob/master/src/full/Agda/Syntax/Parser.hs`

pose is to resolve names, determine operator precedences, and construct a well-formed AST that can be further analyzed by the type checker. Scope checking plays a vital role in transforming the raw syntactic information of the CST into a more meaningful representation that captures the semantic structure of the program.

During scope checking, Agda traverses the CST and maintains a scope environment that keeps track of defined names and their visibility. This environment is used to resolve names by considering the context in which they appear and the available definitions in the current scope. Name resolution is a critical aspect of scope checking, as it disambiguates the meaning of identifiers and enables precise type checking.

In addition to name resolution, scope checking handles Agda's flexible operator syntax. Agda allows users to define custom operators with specific precedence and associativity rules. However, these rules are not explicitly encoded in the CST. Instead, they are stored separately and looked up during scope checking. By consulting the operator precedence table, Agda determines the proper grouping and application of operators, ensuring that the resulting AST accurately reflects the intended semantics of the program. This process of operator precedence resolution is important for constructing a well-formed AST that can be reliably processed by subsequent phases of the compiler.

Again, similar to the problem with Agda's parser, its scope-checker gives up immediately as soon as it finds a single error in name resolution and returns immediately with that single error. One key feature of LSP is Diagnostics which is used to give error and warning information to the client to display inline in the code. It is expected that language implementations will try to give as many helpful error messages as possible using diagnostics. By stopping the analysis right away, Agda gives up a great opportunity of guiding the user towards correctly written code. Another problem is that no partially constructed AST is returned that could be used for implementing LSP features when the code is not correct. This makes it impossible to provide Completions while the user is typing by only relying on the AST.

4.3 Type Checking and Interaction

Agda's type checker plays a crucial role in ensuring the correctness and consistency of Agda programs. It operates on the AST produced by the scope checking phase, traversing the AST to infer types, check type consistency, and generate detailed type information for each program element. The type checker maintains a context that keeps track of the types of variables, functions, and data constructors, as well as the constraints and equalities imposed by the program's logic.

To implement various LSP features, it is essential to be able to resolve imported modules. This can be achieved through Agda's Haskell API using the `scopeCheckModule` function from the `Agda.Interaction.Imports`[2] module. Despite its name, this function not only scope-checks the import but also type-checks it. Consequently, type-checking time remains relevant for implementing LSP, even if we only want to rely on scope-checking information when using Agda's module system.

However, Agda's type-checker is known to be too slow for providing real-time feedback to users (more information about this claim can be found in the Evaluation Chapter 6). To mitigate this issue and keep type-checking times low, Agda employs a caching mechanism. It stores type information for modules and definitions in interface files, allowing for efficient reuse of previously type-checked code. When a module is imported or a definition is referenced, Agda can quickly retrieve the cached type information from the interface files, avoiding the need to re-type-check the dependencies of the program. This mechanism is crucial for providing a responsive and efficient development experience when working with large codebases.

[2] https://github.com/agda/agda/blob/master/src/full/Agda/Interaction/Imports.hs

Nevertheless, the serialization and deserialization of interface files can have performance implications. Loading and saving interface files involves disk I/O operations, which can introduce latency and impact the responsiveness of the development environment. To address these performance issues, it is important to manage Agda's Type Checking Monad (TCM) properly. The simplest way to do this is to retrieve the underlying `TCState` and re-use it whenever we want to check files so that they re-use the loaded interface files instead of reading and deserializing each time from disk.

Agda's type checking process is tightly integrated with its interaction model, which enables interactive development and exploration of Agda programs. The interaction model allows developers to query the type checker, inspect the types of expressions, and incrementally construct proofs and programs using a command-based interface. This interactive nature sets Agda apart from many other programming languages and is a key factor in its appeal to researchers and developers working with dependent types.

At the core of Agda's interaction model lies the TCM, a state monad that encapsulates the context and state of the type checker. The TCM provides a set of APIs and operations for interacting with the type checker, such as retrieving type information, unifying types, and generating proof obligations. These APIs allow developers to interrogate the type checker, ask for the type of an expression, and step through the construction of proofs and programs in a controlled manner.

In the next chapter, we will dive into the details of the LSP implementation for Agda. We will explore how to leverage the information available in the CST and TCM to provide useful language features such as code completion, go-to references and some automated refactorings. We will also discuss the challenges and trade-offs involved.

Chapter 5

Agda LSP Implementation

This chapter presents the implementation of a prototype Language Server Protocol (LSP) for Agda. This implementation leverages a custom scope checker to provide faster and more responsive IDE features, compared to relying solely on Agda's type checker, which is significantly slower but more accurate. By decoupling certain language features from the type checking process, we aim to improve the developer experience and provide useful functionality without the overhead of full type checking.

The implementation focuses on a subset of LSP features that can be realized using scope checking alone, such as semantic highlighting, go to definition, find references, and basic code actions for refactoring implicit variables. While this approach has limitations and cannot provide the full range of features that would be possible with a fully type-checked solution, it offers a pragmatic and efficient way to enhance the Agda development workflow.

Throughout this chapter, we will go into the details of the LSP implementation for Agda. We will discuss the advantages and limitations of the scope checking approach, the choice of programming language and libraries used in the implementation, and the integration with the existing Agda ecosystem, particularly the Visual Studio Code extension agda-mode-vscode. We will also provide an in-depth look at the implementation of specific LSP features, including semantic highlighting, diagnostics, completions, and code actions.

By the end of this chapter, readers will have a comprehensive understanding of the LSP implementation for Agda, its underlying design decisions, and the potential for future improvements and extensions. This work aims to contribute to the growing body of research on language server protocols and their application to dependently typed programming languages, ultimately benefiting the Agda community and the wider field of programming language research.

5.1 Scope Checking Approach

One of the key design decisions in the LSP implementation for Agda was to rely on scope checking rather than full type checking. Scope checking is a static analysis technique that focuses on verifying the visibility and accessibility of identifiers within a program's scope, ensuring that names are properly declared and used in accordance with the language's scoping rules. In contrast, type checking is a more comprehensive process that validates the type consistency and compatibility of expressions and variables throughout the program.

5.1.1 Advantages over Type Checking

The primary advantage of using scope checking over type checking in the context of an LSP implementation is its computational efficiency. Scope checking can be performed relatively quickly by traversing the abstract syntax tree (AST) or concrete syntax tree (CST) of the

program, keeping track of declared identifiers and their scopes. This process does not require the complex type inference and unification algorithms that are typically associated with type checking in languages like Agda.

By leveraging scope checking, we can decouple certain language features from the type checking process and provide useful functionality to the user without the need to wait for the complete type checking of the entire codebase. This is particularly relevant in the context of an IDE, where responsiveness and real-time feedback are crucial for a smooth and productive development experience.

Scope checking allows us to implement features such as semantic highlighting, go to definition, and find references with minimal latency, as these features primarily depend on the visibility and accessibility of identifiers within the program's scope. By avoiding the overhead of full type checking, we can provide these features in real-time, even for large codebases or in the presence of type errors which cause the Agda type checker to halt.

It is important to note that while scope checking offers significant advantages in terms of performance and responsiveness, it does have limitations compared to full type checking. Scope checking alone may not catch certain type-related errors or inconsistencies that would be detected by a complete type checking process. However, the benefits of faster feedback and improved developer productivity often outweigh these limitations, especially in the context of an IDE where the focus is on providing immediate assistance and guidance to the programmer.

In the following subsections, we will explore the specific LSP features that can be implemented using scope checking and discuss their implementation details. We will also address the limitations and trade-offs of the scope checking approach and outline potential future directions for extending the LSP implementation to incorporate type checking where necessary.

5.1.2 LSP Features using Scope Checking

The scope checking approach enables the implementation of several useful LSP features that can significantly enhance the Agda development experience. By leveraging the information obtained from scope analysis, we can provide programmers with real-time assistance and navigation capabilities without relying on the complete type checking of the codebase. In this subsection, we will discuss the most important LSP features that can be implemented using scope checking alone.

Semantic Highlighting

In the context of Agda, semantic highlighting can be implemented using the scope information obtained from the scope checker. By traversing the CST and analyzing the scopes of identifiers, we can determine whether a particular identifier represents a type, function, variable, or other language construct. This information can then be used to apply specific highlighting styles to each identifier, making it easier for programmers to visually distinguish between different elements of their code.

The implementation of semantic highlighting in the Agda LSP involves folding over the CST and checking the scope of each identifier node. If an identifier is found to be defined or used within a certain scope, the corresponding highlighting information is generated and sent back to the client IDE. This process is computationally efficient and can be performed in real-time as the user types, providing instant visual feedback and improving code readability.

Go to Definition and Find References

Both of these features can be implemented effectively using the scope information obtained from scope checking. By maintaining a mapping between identifiers and their declaration

locations, as well as a list of all references to each identifier, the LSP can provide fast and accurate navigation capabilities.

When a user invokes the Go to Definition feature, the LSP looks up the identifier at the current cursor position in the scope map and retrieves the location of its declaration. This information is then used to navigate the user to the appropriate file and position within the codebase. Similarly, for Find References, the LSP consults the reference list associated with the identifier and returns a list of all locations where the identifier is used.

Code Actions for Implicit and Unused Arguments

Code actions are a powerful LSP feature that enables the language server to offer context-sensitive suggestions and refactorings based on the user's current code. In the case of Agda, one particularly useful code action is the ability to manage implicit arguments in function declarations and applications.

Implicit arguments are a common feature in dependently typed languages like Agda, allowing programmers to omit certain arguments when they can be inferred from the context. However, managing implicit arguments can sometimes be tedious, especially when dealing with complex function signatures or when refactoring code.

The Agda LSP implementation leverages scope checking to provide code actions for implicit argument management. By analyzing the type signatures of functions and the patterns of their applications, the LSP can offer suggestions such as making an implicit argument explicit or removing unused implicit arguments altogether.

To implement these code actions, the LSP keeps track of the type signatures of functions and data constructors during scope analysis. By examining the names and positions of implicit arguments in these signatures, the LSP can determine which arguments are currently implicit and which ones are explicitly provided in function applications.

When the user invokes the code action feature, the LSP analyzes the current context and generates a list of applicable actions based on the implicit argument information. These actions can include inserting missing implicit arguments and removing unused implicits or explicits. The user can then select the desired action, and the LSP will automatically apply the corresponding refactoring to the code.

By providing these implicit argument code actions, the Agda LSP significantly improves the developer experience, reducing the manual effort required to manage implicits and making the code more readable and maintainable.

Diagnostics for Unused Symbols

Unused symbol warnings are a common feature in many programming languages, alerting developers to variables, functions, or other identifiers that are declared but never used in the code. These warnings can help improve code quality, reduce clutter, and catch potential bugs early in the development process.

The Agda LSP implementation leverages the scope information obtained from scope checking to generate unused symbol diagnostics. During scope analysis, the LSP keeps track of all declared identifiers and their usage within the code. By comparing the set of declared identifiers with the set of actually used identifiers, the LSP can determine which symbols are unused.

When the client IDE requests diagnostics for a file, the LSP traverses the scope representation and generates warnings for each unused symbol it encounters. These warnings include the location of the unused symbol (i.e., the range in the source file where it was declared) and a descriptive message indicating that the symbol is unused.

The client IDE can then display these warnings to the user, typically in the form of colored squiggles or underlines in the code editor. Users can hover over the warnings to see

the detailed message and take appropriate action, such as removing the unused symbol or updating the code to use it.

It's worth noting that the scope checking approach to unused symbol diagnostics may have some limitations compared to a full type-checking solution. For example, it may not be able to detect unused symbols that are only referenced in dead code or unreachable branches. However, for most practical purposes, the scope-based unused symbol diagnostics provide a valuable tool for improving code quality and catching potential issues early in the development process.

5.1.3 Limitations and Trade-offs

While the scope checking approach offers significant benefits in terms of performance and responsiveness, it is important to acknowledge its limitations and the trade-offs involved in using it as the basis for an LSP implementation.

Limited Range of Features and Diagnostics

One of the main limitations of scope checking is that it may not be able to provide the full range of features and diagnostics that would be possible with a complete type checking solution. Scope checking focuses primarily on the visibility and accessibility of identifiers, but it does not perform the deep semantic analysis and type inference that are necessary for catching more subtle type-related errors or providing more advanced code suggestions.

For example, scope checking alone may not be sufficient to detect issues such as type mismatches, invalid function applications, or violations of Agda's complex type-level constraints. These kinds of errors require a more thorough understanding of the type system and the relationships between different type expressions, which can only be achieved through full type checking. So error diagnostics as typically expected in full LSP implementations will not be available.

Additionally, certain LSP features, such as type-aware code completion or type-driven refactorings, may be more challenging to implement using scope checking alone. Also, Hover information, when the User holds the cursor over an identifier, will not be complete. It will at most show what kind of symbol it is (Constructor, Function, etc.) but it will not show the actual type of the symbol.

Maintenance and Compatibility

Another limitation of the custom scope checking approach is that it may require more manual effort to maintain and update as the Agda language evolves. While the scope checker can be implemented as a standalone component, it still needs to be kept in sync with changes to the Agda parser and abstract syntax tree. This may involve updating the scope checker to handle new language features, syntax extensions, or changes to the scoping rules. This limitation can be easily addressed by some simple modifications to the existing Agda scope checker, so that instead of using a custom implementation, the main Agda scope checker is used.

Balancing Performance and Functionality

Despite these limitations, the scope checking approach still provides a valuable and pragmatic solution for enhancing the Agda development experience. By focusing on a core set of features that can be implemented efficiently using scope information, the LSP can offer significant benefits to programmers, such as real-time feedback, navigation, and basic refactoring support. This is in contrast to the massive amount of effort that would be required to make the existing type checker fast enough to run in real time by using techniques such as incremental type checking.

In the meantime, the scope checking approach offers a balance between performance, responsiveness, and functionality, enabling Agda programmers to benefit from a more interactive and supportive development environment. While it may not provide the full range of features possible with a type-checked solution, it represents a significant step forward in terms of tooling and developer experience for the Agda community.

5.2 Implementation Details

In this section, we will discuss the implementation details of the Agda LSP server and its key components and design decisions. Subsection 5.2.1 will explore the rationale behind selecting Haskell as the primary language for the Agda LSP implementation. Subsection 5.2.2 will focus on the custom scope checker, the core component of the implementation that enables fast and responsive IDE features by operating on Agda's concrete syntax tree (CST). The integration of the Agda LSP with the existing Visual Studio Code extension, agda-mode-vscode, and the challenges encountered during this process will be discussed in Subsection 5.2.3. Finally, Subsection 5.2.4 will explore the implementation of the implicit argument management feature, which aims to enhance the user experience by providing algorithms for generating insertions and removals of implicit arguments.

5.2.1 Choice of Programming Language

The choice of programming language is a critical decision in the implementation of any software project, and the Agda LSP is no exception. Given the nature of the project and its close integration with the Agda ecosystem, Haskell was selected as the primary language for the LSP implementation.

Haskell is a statically-typed, purely functional programming language that is well-suited for developing compilers, interpreters, and other language-related tools. It offers a rich type system, powerful abstractions, and a strong emphasis on correctness and reliability. These features make Haskell an ideal choice for implementing the Agda LSP, which requires precise and efficient manipulation of syntax trees, scopes, and other language-related data structures.

One of the main reasons for choosing Haskell is its close relationship with Agda itself. Agda is implemented in Haskell, and many of its core components, such as the parser and the type checker, are written in Haskell. By using the same language for the LSP implementation, we can leverage existing Agda libraries and infrastructure, reducing the development effort and ensuring better integration with the Agda ecosystem.

Haskell has a mature and well-established ecosystem, with a wide range of libraries and tools that can be used in the development of the Agda LSP. Of particular relevance is the `lsp` library, which provides a strongly-typed framework for implementing Language Server Protocol servers in Haskell. It is the same library used in the production grade LSP servers for Haskell and Futhark, which shows it is well maintained. It provides a set of pre-defined data types and functions that correspond to the various LSP messages and capabilities, making it easier to implement compliant LSP servers.

5.2.2 Custom Scope Checker

The Agda LSP implementation relies on a custom scope checker to enable fast and responsive IDE features. While Agda already includes a scope checker as part of its compiler infrastructure, it has some limitations that make it less suitable for direct use in an LSP context. Specifically, the existing parser and scope checker stops as soon as it encounters a single error, without attempting any recovery. This makes it unusable when the code finds itself in

incorrect intermediate stages while being written by the user, which is when we need LSP features the most.

To address these limitations, we implemented a custom scope checker tailored to the needs of the LSP. The custom scope checker operates on Agda's concrete syntax tree (CST) and recursively builds up a hierarchical representation of the program's scope. At each node in the CST, the scope checker maintains a mapping of identifiers to their corresponding symbol information, such as their kind (e.g., type, function, variable), location, and any associated metadata.

One of the challenges in implementing the custom scope checker was handling Agda's rich syntax and scoping rules. Agda supports a wide range of language constructs, including dependent types, pattern matching, mixfix operators, and implicit arguments. To accurately capture the scoping behavior of these constructs, the scope checker needs to recursively traverse the CST and update the scope mappings accordingly.

For example, when encountering a function definition, the scope checker creates a new scope for the function body and adds any explicitly declared arguments to the scope mapping. It then recursively processes the function body, which may introduce additional local scopes for constructs like pattern matching or let-expressions. As the scope checker traverses back up the CST, it appends the local scopes to their parent scopes to create a unified view of the program's scope hierarchy.

Another challenge in implementing the custom scope checker was dealing with the limitations of working directly with the CST. Unlike the abstract syntax tree (AST), which is a more structured representation of the program, the CST includes additional syntactic details and may not always provide a clean separation between different language constructs. This can make it more difficult to accurately track scoping information, particularly for complex constructs like mixfix operators.

Despite these challenges, the custom scope checker provides an efficient foundation for implementing LSP features in Agda. By maintaining a standalone representation of the program's scope, the scope checker enables fast queries and manipulations of symbol information, without the need for full type checking. This allows the LSP to provide responsive feedback and navigation capabilities, even in the presence of parse, scope and type errors.

It's worth noting that the custom scope checker is not intended to replace Agda's existing scope checker. The custom scope checker was just necessary to bypass the lack of error recovery on the existing Agda scope checker. The aim is to show that through scope checking alone many important LSP features can be implemented which should hopefully inspire the Agda maintainers to fix these small limitations so that a future LSP implementation based on scope checking can use the canonical Agda implementation.

5.2.3 Integration with agda-mode-vscode

To make the Agda LSP implementation accessible to users, we integrated it with the existing Visual Studio Code extension for Agda, called agda-mode-vscode[1]. agda-mode-vscode brings Agda's Emacs mode functionality into VSCode. It's important to note that prior to this integration, agda-mode-vscode did not use LSP for its functionality. Instead, it forwards Emacs style commands to the Agda executable in a similar fashion to the original agda-mode.

Integrating the Agda LSP with agda-mode-vscode involved several steps. We updated the extension's code to establish a connection with the Agda LSP server and to handle the exchange of LSP messages. This involved using Visual Studio Code's built-in LSP client library to send requests and notifications to the LSP server, as well as processing the responses and updating the editor's user interface accordingly.

[1] https://github.com/banacorn/agda-mode-vscode

One of the challenges in integrating the Agda LSP with agda-mode-vscode was ensuring a smooth interoperability between the LSP features and the existing Emacs mode functionality. Agda-mode-vscode relies heavily on the Emacs mode for certain features, such as interactive development and goal-directed programming. To maintain compatibility and provide a seamless user experience, we implemented the LSP integration in a way that preserves the existing Emacs mode features while augmenting them with the additional capabilities provided by the LSP.

The integration of the Agda LSP with agda-mode-vscode significantly enhances the development experience for Agda programmers using Visual Studio Code. And makes it much more likely to be incorporated into user's workflows. By combining the power of the LSP with the rich functionality of the existing Emacs mode, agda-mode-vscode is now enhanced with code completion, real time syntax highlighting, diagnostics and some refactorings while still maintaining the familiar Emacs mode with its powerful capabilities.

5.2.4 Implicit Argument Management

Implicit argument management is a key feature of our Agda Language Server, aimed at enhancing the user experience and reducing the cognitive burden on developers. The algorithm for handling implicit arguments consists of two main components: generating insertions and generating removals.

Intermediate Representation: IPattern

To facilitate the processing of Agda patterns, the algorithm first converts them into an intermediate representation called `IPattern`. This representation captures various forms of patterns, including those involving implicit arguments, such as hidden patterns and patterns within parentheses. Figure 5.1 shows the definition of the `IPattern` data type. The key difference between the CST patterns is that variables contain a symbol identifier that is used to lookup whether a that variable is unused. And functions also contain a list of `SigParam` which holds information on what kinds of parameters and how many this function receives as input. This information is used in the matching stage when trying to find which patterns match which parameters.

```
data IPattern = IApp LSP.Range (Maybe SymId) [SigParam] [IPattern]
              | IHidden LSP.Range (Maybe String) IPattern
              | IVar LSP.Range (Maybe String) (Maybe SymId)
              | IOther LSP.Range
              | IParen LSP.Range IPattern
```

Figure 5.1: IPattern data type. Each pattern contains a range representing its position in the source file, which is important when generating the Code Actions for removing and inserting implicit arguments since a Code Action needs to know in which range its edit will be inserted. It also contains symbol identifiers to simplify looking up whether a symbol is unused or not in the matching stage.

The conversion from Agda patterns to `IPattern` is performed by a function that utilizes the provided scope to resolve symbols and handle different pattern types.

Generating Insertions

To generate insertions, the algorithm computes all possible insertions of implicit arguments that are not explicitly stated in the code. It does so by recursively processing the `IPattern` and determining valid insertion points based on the signature parameters and the existing

patterns. A matching function plays a crucial role in aligning the patterns with their corresponding parameters, enabling accurate identification of missing implicit arguments.

The `Insertion` data type, shown in Figure 5.2, represents the specific details of an insertion, including the range in the source code, the name of the argument, and any necessary additional information.

```
data Insertion = Insertion LSP.Range String (Maybe String) (Maybe LSP.Range)
```

Figure 5.2: Insertion data type representing an implicit parameter that is not made explicit and could be automatically made explicit through Code Actions.

The first parameter is the edit range where this variable should be inserted if the Code Action is executed. The second parameter is the name of the variable to insert, which is used to display the name of the variable to users in the Code Actions context menu and also as the string that is actually inserted into the program.

The third parameter is an optional string that contains the name of the inserted implicit argument when it needs to be inserted in a position that is not immediately after the next implicit parameter. For example, consider the following Agda code:

```
add : {a b : Nat} -> Nat
add = ?
```

In this case, we have the function clause `add` with implicit parameters `a` and `b`. If we insert `b` without first inserting `a`, we must give a name to `b` so that Agda knows which implicit to match. The resulting code after automatically inserting `b` would be:

```
add : {a b : Nat} -> Nat
add {b = b} = ?
```

The last optional range in the `Insertion` data type represents the positions of parentheses that must be inserted if this implicit is inserted. This is necessary when, for example, a constructor has implicit parameters, and none of them are made explicit yet. In such cases, if any implicit is inserted, the constructor on the pattern would need to be wrapped in parentheses.

Here's an example to illustrate this scenario:

```
data MyData : Set where
  Data : {a : Nat} -> MyData

foo : MyData -> Nat
foo Data = ?
```

If we want to insert the implicit parameter `a` in the constructor `Data` that is pattern matching the first parameter of `foo`, after applying the insertion the resulting code would be:

```
foo : MyData -> Nat
foo (Data {a}) = ?
```

Notice that the constructor `Data` is now wrapped in parentheses to accommodate the inserted implicit parameter. If we did not insert the parenthesis Agda's parser would interpret the implicit argument to belong to the function instead of the constructor.

Generating Removals

Generating removals follows a similar approach to generating insertions. The algorithm identifies unused implicit arguments within a given scope and position in the source code. It converts the pattern to an `IPattern` and uses the unused symbols information from the scope to determine which symbols are unused. The algorithm then generates a list of `Removal` items,

specifying the range and details of the implicit arguments to be removed. Figure 5.3 shows the definition of the `Removal` data type.

```
data Removal = Removal String Bool LSP.Range (Maybe (LSP.Range, String)) (Maybe LSP.Range)
```

Figure 5.3: Removal data type representing an unused variable that can be automatically removed through Code Actions.

The `Removal` data type represents an unused variable that can be automatically removed through Code Actions. The first parameter is the name of the variable, used to display to the user which variable will be removed.

The second parameter is `True` when the variable can be fully removed. For example, when removing an unused explicit variable, we cannot remove it fully since that would change the semantics of the program. Instead, we replace it with an underscore. This is also the case when an implicit parameter is pattern matched to a constructor and there is a variable like `{suc a}` where `a` is unused. Here, we can't remove `a` or we will no longer be matching on the `suc` constructor, we must insert an underscore instead.

The third parameter contains the range in the source file of the variable, which is used to provide an edit range for the LSP command to know what to remove.

The fourth parameter optionally contains the name and where to insert this name that must be given to the next implicit variable in the case where we remove a positional implicit variable and the one to the right is also positional and requires a name to keep referring to the same one. Consider the following example:

```
add : {a b : Nat} -> Nat
add {a} {b} = ?
```

If we remove `a` because it is unused, we will need to give a name to `b` so that it keeps referring to the same variable. The result after applying the refactoring will be:

```
add : {a b : Nat} -> Nat
add {b = b} = ?
```

The final parameter is an optional range that contains the positions of parentheses that will be redundant after removing the variable. For instance, consider a constructor that has an only implicit argument that is made explicit but is unused:

```
data MyData : Set where
  Data : {a : Nat} -> MyData

foo : MyData -> Nat
foo (Data {a}) = ?
```

When we remove the unused implicit argument `a` from the first pattern of the constructor `Data` in `foo`, we should also remove the parentheses around the constructor as they become unnecessary:

```
foo : MyData -> Nat
foo Data = ?
```

Removing these parenthesis is not strictly necessary as semantically the programs would remain the same with or without parenthesis there. But it does reduce unnecessary clutter in the syntax.

Auxiliary Functions

To find the right location to search for implicit arguments, the Language Server exposes two main functions. These functions find the relevant clause containing the cursor position and

return a list of implicit arguments that can be inserted or variables that can be removed, respectively. They utilize a function to locate the appropriate pattern within the module.

The implicit argument management algorithm relies on several auxiliary functions and data types to facilitate its operations. These functions handle complex nesting of patterns by recursively processing sub-patterns and applying the necessary logic for insertions and removals. They take into account hidden patterns, patterns within parentheses, and other edge cases to ensure comprehensive coverage of all potential modifications.

In conclusion, the implicit argument management algorithm in our Agda Language Server provides a robust and efficient way to handle the insertion and removal of implicit arguments in Agda source code. By utilizing an intermediate representation, recursively processing patterns, and leveraging auxiliary functions and data types, the algorithm ensures comprehensive coverage of all potential modifications, enhancing the user experience and streamlining the development process.

Chapter 6

Evaluation

In this chapter, we evaluate our approach, focusing on the performance and feature completeness of our custom CST scope checker LSP implementation. We begin by detailing our performance evaluation in Section 6.1, where we compare the speed and efficiency of our approach against the existing Agda type checker and other relevant components. Following this, we delve into a feature evaluation in Section 6.2, assessing the capabilities of our LSP implementation against a hypothetical full LSP implementation and the well-established Agda Emacs mode.

6.1 Performance Evaluation

In this section, we compare the performance of the Agda CST Parser, Agda Scope Checker, Agda Type Checker, and the custom CST scope checking technique developed for the LSP implementation for Agda. The purpose of this evaluation is to assess the feasibility of providing real-time feedback to users while they edit Agda code. We describe the methodology and test setup used to evaluate these components, present the results in a clear and visually appealing format, and analyze the findings to discuss the performance differences and their implications.

6.1.1 Methodology and Test Setup

To evaluate the performance of the Agda CST Parser, Agda Scope Checker, Agda Type Checker, and the custom scope checking technique, we will be testing on the source code of the Agda standard library. The Agda standard library is an extensive collection of Agda modules that cover a wide range of mathematical concepts and programming constructs. It includes modules for basic data types, data structures, algorithms, and various branches of mathematics such as algebra, topology, and category theory. The diversity and complexity of the modules in the standard library make it an ideal dataset for testing the performance of the aforementioned components under various scenarios, ensuring that the results are representative of real-world Agda development.

The tests were performed on all files in the Agda standard library to better understand the worst-case and average-case performance. This approach allows us to identify potential performance bottlenecks and edge cases that may arise during the development process. Before measuring the time taken for each phase of the type checker on a specific file, we first ran the Agda type checker at least once on all files in the library. This initial type check ensures that all dependencies of the file being tested are pre-checked, and their types are cached in Agda interface files. By doing this, we simulate a realistic editing scenario where the user has already type-checked all imported files, and the results are readily available in the cache. This step is important because it allows us to isolate the performance measurement of each

phase for the specific file being tested, without including the time spent on checking its imports. Consequently, the measured time for each phase reflects only the time taken to process the current file, providing a more accurate representation of the performance experienced by users while editing Agda code in an IDE. This approach ensures that the performance measurements are not skewed by the time spent on checking dependencies, which would have already been cached in a typical editing session.

The measurements were performed 8 times per file to ensure the reliability of the results and to account for any potential variations in performance due to external factors such as system load or cache state. The custom scope checker's benchmark was measured using "getCPUTime" from "System.CPUTime", which provides high-resolution CPU time measurements. The rest of the components were timed using the built-in Agda profiler, which offers detailed information about the time spent in each phase of the Agda compilation process.

All benchmarks were conducted using Haskell GHC 9.2.8 on Ubuntu 22.04 LTS, with Agda 2.6.3. The tests were performed on an HP ZBook Studio G5 with 32.0 GiB of memory and an Intel® Core™ i7-8750H CPU @ 2.20GHz × 12 processor.

6.1.2 Results

The results of the performance evaluation are presented in Table 6.1, Figure 6.2 and Figure 6.3.

Table 6.1 shows the execution times for each component on a selection of Agda files from the standard library. The columns represent the file name, line count in the file, Agda CST Parser, Agda Scope Checker, Agda Type Checker, and custom scope checking execution times, while the rows represent the individual test files. The files manually selected for this table aim to showcase the performance of the components across a diverse set of Agda modules, ranging from basic data types to more complex mathematical concepts.

Figure 6.2 presents a bar graph comparing the average execution times of each phase across all files in the Agda standard library. The x-axis represents the phases (Agda CST Parser, Agda Scope Checker, Agda Totality Checker (Termination + Positivity), Agda Type Checker, and custom scope checking), while the y-axis represents the average execution time in milliseconds.

The average times for each of these phases can be a bit misleading as the higher percentiles for each phase can take a lot longer. Figure 6.3 shows a box plot of the same information as the previous figure.

File name	Lines	Parsing	Scoping	Typing	Custom Scoping
Data.Nat.Base	240	5.70 ms	5.56 ms	11.32 ms	4.28 ms
Data.List.Base	486	23.36 ms	22.38 ms	69.77 ms	12.67 ms
Algebra.Solver.Ring	551	76.80 ms	246.33 ms	234.61 ms	15.98 ms
Algebra.Properties.Group	138	8.57 ms	6.75 ms	25.18 ms	3.53 ms
Function.Bijection	127	5.33 ms	3.29 ms	20.79 ms	2.01 ms
Category.Functor	46	3.24 ms	2.01 ms	5.11 ms	0.86 ms
Relation.Binary.PropEq	146	6.89 ms	5.23 ms	22.16 ms	4.23 ms
Reflection	235	2.91 ms	2.20 ms	3.81 ms	2.87 ms

Table 6.1: Line count and average execution times for each component on selected Agda files from the standard library.

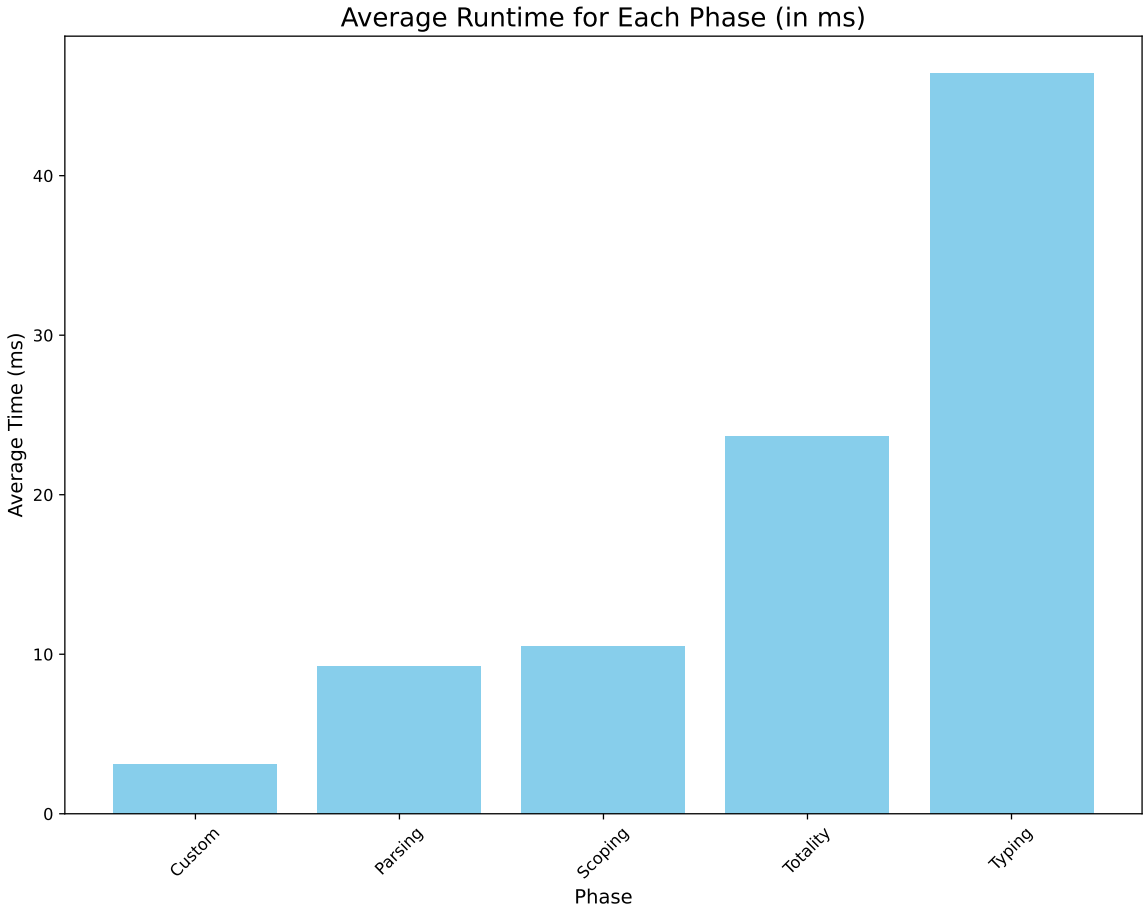


Figure 6.2: Bar graph comparing the average execution times of each phase across all files in the Agda standard library.

6.1.3 Analysis and Discussion

The results show that the Agda CST Parser and the custom CST scope checker performs exceptionally well, with an average execution time of under 10 milliseconds per file. This is a promising finding, as it suggests that the technique developed in this paper can provide real-time feedback to users without introducing significant latency. Research on web user experience (UX) suggests that waiting times between 0 and 100ms feel instantaneous to users, while waiting times between 100ms and 300ms are noticeable but not perceived as slow (Arapakis, Bai, and Cambazoglu 2014). Although this research focuses on web UX, we can extrapolate these findings to the optimal response times for real-time feedback in IDEs. The custom CST scope checker's performance falls well within the instantaneous feedback range, making it suitable for integration into an interactive development environment.

Looking at the percentiles in Figure 6.3 we see a different picture. The custom scope checker performs exceptionally well in all percentiles. The 95th percentile for the Parsing and Scoping phases is still under 100ms which suggests that using the Agda Parser and Scope Checker would be feasible for giving real time feedback on an LSP implementation. The 99th percentile of runtimes is large for the Agda phases. For Parsing it is still under 100ms, but Scoping exceeds this and Typing far exceeds the 1000ms mark making it unusable for real time user feedback.

The difference between the 95th and 99th percentile is high: this means that there are some outliers that take extremely long times go through each phase. In Table 6.4 we can see the top 5 files sorted by decreasing Scoping runtimes. And we can clearly see that they are

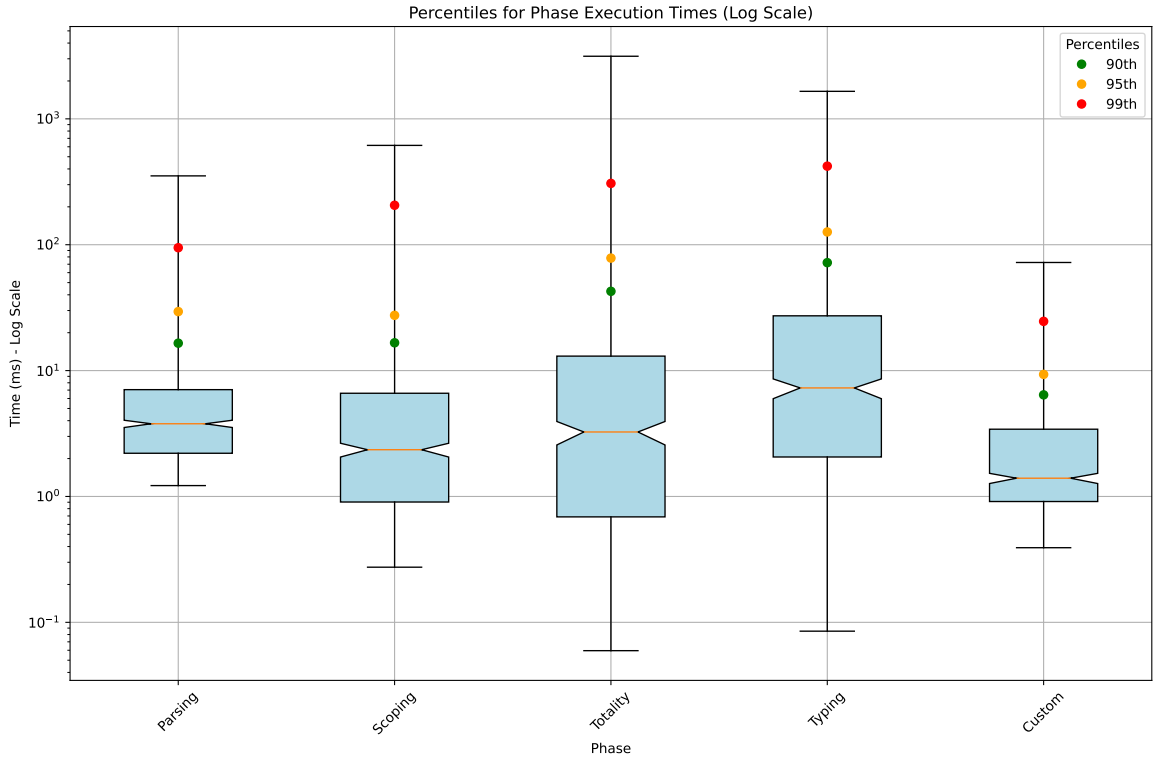


Figure 6.3: Box plot showing the distribution of execution times for each phase on each file of the Agda standard library, with whiskers extending to the minimum and maximum values to include the full data range.

all quite large files, with around 1000 lines or more. This makes sense, as larger files have a larger syntax tree and require more operations. And it also aligns with the common advice when writing Agda to keep your files small so that they can get easily cached by the Agda compiler. Smaller files get processed much faster. In Figure 6.5 we can see all phase times plotted against the line count of each file in the x-axis. We can see that there are no files under 260 lines that take more than 100ms to scope check. This means that it is definitely feasible to use the fully implemented and correct Agda Scope Checker to implement an LSP.

File name	Lines	Parsing	Scoping	Typing	Custom Scoping
Data.Integer.Properties	2501	352.29 ms	615.34 ms	757.37 ms	72.25 ms
Data.Fin.Subset.Properties	869	102.72 ms	509.03 ms	386.35 ms	21.35 ms
Data.Nat.Properties	2395	269.73 ms	465.42 ms	419.12 ms	54.18 ms
Data.Fin.Properties	962	71.26 ms	396.83 ms	229.51 ms	27.02 ms
Data.Nat.Binary.Properties	1530	218.05 ms	311.86 ms	372.06 ms	43.54 ms

Table 6.4: Line count and execution times for each component on the files of the Agda standard library that were slowest to scope check.

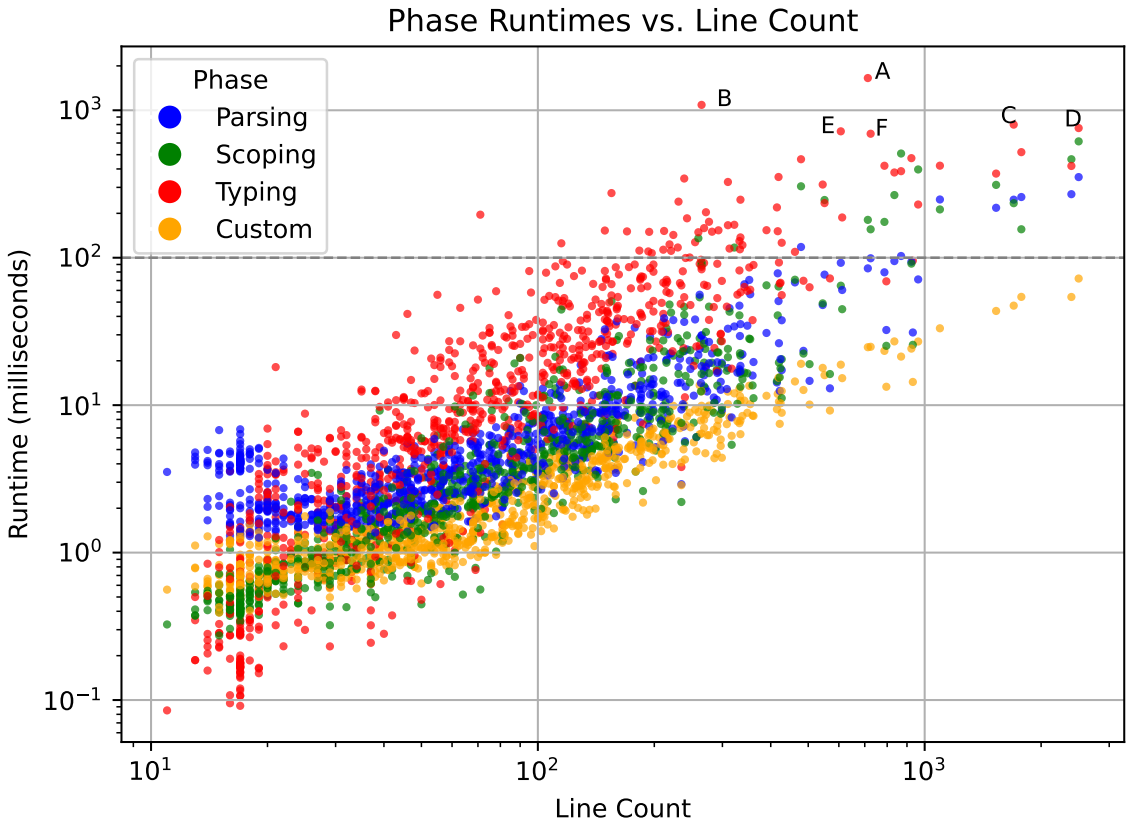


Figure 6.5: The graph depicts how runtime varies with line count for key phases in the Agda compiler: Parsing, Scoping, Typing, and Custom. Each point represents a file, showing its line count and phase runtime. Colors distinguish phases for easy comparison, while a dashed line at 100ms denotes the ideal maximum runtime for real time interactivity (Arapakis, Bai, and Cambazoglu 2014). Further information about labelled points can be seen in Figure 6.6.

Label	File name	Lines	Parsing	Scoping	Typing	Custom Scoping
A	Data.List.Relation. Binary.Sublist. Heterogeneous. Properties	713	84.69 ms	180.60 ms	1654.21 ms	24.73 ms
B	Data.Tree.AVL. Indexed.Relation. Unary.Any.Properties	265	29.80 ms	88.94 ms	1087.88 ms	12.42 ms
C	Data.Rational. Properties	1699	247.53 ms	234.33 ms	799.32 ms	47.31 ms
D	Data.Integer. Properties	2501	352.29 ms	615.34 ms	757.37 ms	72.25 ms
E	Data.List.Relation. Binary. BagAndSetEquality	607	92.37 ms	64.60 ms	719.87 ms	18.82 ms
F	Data.List.Relation. Unary.Any.Properties	725	99.14 ms	155.86 ms	693.33 ms	24.88 ms

Table 6.6: Line count and execution times for each component on the files of the Agda standard library that were slowest to type-check.

One limitation of this benchmark is that the current implementation of the custom scope checker is not complete and does not traverse all Agda declarations. As a result, the performance numbers reported in these benchmarks may be slightly lower than what would be observed if all declarations were fully implemented. However, even if the custom scope checker were to take twice as long to execute, its performance would still be well within the acceptable range of under 100ms for providing real-time feedback to users.

6.2 Feature Completeness

In this section, we evaluate the feature completeness of the custom CST scope checker LSP implementation, comparing it to a hypothetical full LSP implementation and the existing Agda Emacs mode. The comparison is based on various categories of features, including editor integration, code navigation, code assistance, refactoring, error handling, code formatting, and Agda-specific features.

Tables 6.7 through 6.13 provide a detailed breakdown of the feature support for each implementation across these categories. The tables use a color-coding scheme to indicate the level of support: green for comprehensive support, yellow for partial or limited support, red for no support, and blue for exceptional support.

6.2.1 Editor Integration

The custom CST scope checker LSP offers basic syntax highlighting based on scope analysis, while a full LSP implementation would provide more comprehensive highlighting using type information as well. The Agda Emacs mode provides syntax highlighting using Emacs' font-lock mode. Code folding is not supported in the custom implementation but could be added, while both the full LSP and Emacs mode support folding based on the semantic structure or using Emacs' outline-minor-mode, respectively. Unicode input methods are supported via the agda-vscode extension in the custom implementation, depend on the editor's capabilities in a full LSP, and are well-supported in the Emacs mode.

Editor Integration	**Custom CST scope checker LSP**	**Full LSP**	**Agda Emacs mode**
Syntax Highlighting	Basic syntax highlighting based on scope analysis.	Comprehensive syntax highlighting using semantic information.	Syntax highlighting with Emacs' font-lock mode.
Code Folding	Could be supported but not implemented.	Code folding based on semantic structure.	Code folding using Emacs' outline-minor-mode.
Unicode Input Method	Supported via agda-vscode extension.	Depends on the editor's Unicode input capabilities.	Agda-specific Unicode input method for math symbols.

Table 6.7: Editor Integration Feature Comparison

6.2.2 Code Navigation

The custom CST scope checker LSP and a full LSP implementation both support "Go to Definition" and "Find References" using scope information and semantic analysis, respectively. The Agda Emacs mode supports "Go to Definition" and "Find References" using Emacs tags. Document Symbols could be supported using CST and scope information in the custom implementation, but they were not implemented. In a full LSP implementation they would be supported, while the Emacs mode provides partial support using Emacs imenu. Workspace

Symbols and Document Links are not supported in the custom implementation or the Emacs mode, while a full LSP could support these features.

Code Navigation	**Custom CST scope checker LSP**	**Full LSP**	**Agda Emacs mode**
Go to Definition	Supported using scope information.	Supported using semantic analysis.	Supported using Emacs tags.
Find References	Supported using scope information.	Supported.	Supported using Emacs tags.
Document Symbols	Could be supported but not implemented.	Supported using semantic analysis.	Partial support using Emacs imenu.
Workspace Symbols	Could be supported but not implemented.	Supported using workspace-level analysis.	Not supported. This is an LSP specific feature.
Document Links	Not supported.	Could support for certain link types. But the utility for Agda is not clear.	Not supported.

Table 6.8: Code Navigation Feature Comparison

6.2.3 Code Assistance

The custom CST scope checker LSP supports auto-completion using scope information, while a full LSP would also use type information for more comprehensive completions. The Emacs mode could potentially support auto-completion using Emacs' completion mechanisms. Hover information has limited support in the custom implementation, showing only symbol information such as if it is a constructor or a type, while a full LSP would provide comprehensive hover information with type and documentation. The Emacs mode could support hover using Agda's type checker. Signature Help is not supported in any of the implementations due to the difficulty of integrating it with ML-like syntaxes.

Code Assistance	**Custom CST scope checker LSP**	**Full LSP**	**Agda Emacs mode**
Auto-completion	Supported using scope information.	Supported using scope and type information.	Could be supported using Emacs completion mechanisms.
Hover Information	Limited support for showing some symbol information.	Comprehensive hover information with type and documentation.	Could be supported using Agda's type checker.
Signature Help	Not supported due to lack of type information.	Not supported due to difficulty with integrating for ML-like syntaxes.	Not supported due to difficulty with integrating for ML-like syntaxes.

Table 6.9: Code Assistance Feature Comparison

6.2.4 Refactoring

The custom CST scope checker LSP supports Rename Refactoring using scope information, although it might not be fully correct in all cases. A full LSP implementation would support renaming using Agda's scope checker, while the Emacs mode has partial support using

search and replace. Code Actions have limited support in the custom implementation for certain actions, while a full LSP would provide extensive code actions based on semantic analysis. The Emacs mode offers some code actions through Emacs commands.

Refactoring	**Custom CST scope checker LSP**	**Full LSP**	**Agda Emacs mode**
Rename Refactoring	Supported using scope information, although might not be fully correct.	Supported using Agda's scope checker.	Partial support using search and replace.
Code Action	Limited support for certain code actions.	Extensive code actions based on semantic analysis.	Some code actions available through Emacs commands.

Table 6.10: Refactoring Feature Comparison

6.2.5 Error Handling

The custom CST scope checker LSP provides limited error diagnostics for scope-related errors and unused variables, while a full LSP implementation would offer comprehensive error diagnostics using type checking. The Agda Emacs mode supports error diagnostics using Agda's interactive mode. Code Lens is not supported in the custom implementation or the Emacs mode but could be supported in a full LSP if implemented properly for Agda. Document Highlighting could be supported using scope information in the custom implementation but was not implemented in time. On a full LSP implementation it could definitely be implemented using scope information. Emacs does not have this feature.

Error Handling	**Custom CST scope checker LSP**	**Full LSP**	**Agda Emacs mode**
Error Diagnostics	Limited to scope-related errors and unused variables.	Comprehensive error diagnostics using type checking.	Supported using Agda's interactive mode.
Code Lens	Could be supported but not implemented.	Support for code lens features if implemented properly for Agda.	Not supported.
Document Highlighting	Could be supported but not implemented.	Supported using semantic analysis.	Not a standard Emacs feature.

Table 6.11: Error Handling Feature Comparison

6.2.6 Code Formatting

Code formatting is not supported in the custom CST scope checker LSP due to limitations in parsing. It could be supported in a full LSP implementation or in the Emacs mode but as of the time of this writing Agda does not have a standard code formatter.

6.2.7 Agda-Specific Features

The custom CST scope checker LSP and a full LSP implementation do not support the advanced interactive editing features that are the hallmark of the Agda Emacs mode, such as interactive editing, goal-directed programming, case splitting, hole filling, and proof search. While a full LSP could potentially support these features to some extent, depending on the

Code Formatting	Custom CST scope checker LSP	Full LSP	Agda Emacs mode
Code Formatting	Not supported due to limitations in parsing.	Not supported due to Agda not having a code formatter rules.	Not supported due to Agda not having a code formatter rules.

Table 6.12: Code Formatting Feature Comparison

editor's capabilities, the Emacs mode provides exceptional, well-integrated support that sets the standard for interactive development in Agda. It could be possible to implement these features in LSP through Code Actions or Code Lenses or perhaps through extensions to LSP such as SLSP discussed in Section 3.3. But further research is required on this topic.

Agda-Specific Features	Custom CST scope checker LSP	Full LSP	Agda Emacs mode
Interactive Editing	Not supported.	Partial support depending on the editor's capabilities.	Extensive support using Agda's interactive mode.
Goal-Directed Programming	Not supported.	Partial support depending on the editor's capabilities.	Well-integrated support using Agda's interactive mode.
Case Splitting	Not supported.	Partial support depending on the editor's capabilities.	Well-integrated support using Agda's interactive mode.
Hole Filling	Not supported.	Partial support depending on the editor's capabilities.	Well-integrated support using Agda's interactive mode.
Proof Search	Not supported.	Partial support depending on the editor's capabilities.	Well-integrated support using Agda's interactive mode.

Table 6.13: Agda-Specific Feature Comparison

In summary, the custom CST scope checker LSP offers a solid foundation for basic editor integration, code navigation, and limited code assistance and refactoring. However, it lacks the more advanced features that would be possible with access to full type information, such as comprehensive error diagnostics, type-aware completions, and extensive code actions. The Agda Emacs mode remains the gold standard for Agda-specific features, particularly in terms of interactive editing and proof assistance, which are not yet well-supported in LSP implementations.

Chapter 7

Discussion

This thesis aimed to explore the potential of leveraging the Language Server Protocol (LSP) to enhance the development experience for dependently typed languages, focusing specifically on Agda. The main goal was to investigate how LSP features could be implemented efficiently using scope checking, rather than relying solely on full type checking. The research findings demonstrate that scope checking can serve as a foundation for providing fast and responsive LSP features, offering significant improvements in terms of real-time feedback and interactivity.

The significance of this research lies in its contribution to the growing body of knowledge on language server protocols and their application to dependently typed programming languages. By showcasing the feasibility and benefits of using scope checking for LSP features, this thesis opens up new possibilities for enhancing the development experience in Agda and other similar languages. The insights gained from this research can inform future efforts to improve tooling support and promote the adoption of dependently typed languages in both academic and industrial settings.

7.1 Scope Checking as a Foundation for LSP Features

One of the key findings of this research is that scope checking can serve as a powerful foundation for implementing LSP features in Agda. By leveraging the information obtained from scope analysis, it is possible to provide fast and responsive IDE features without relying on the computationally expensive process of full type checking.

The primary advantage of using scope checking for LSP features is its computational efficiency. Scope checking can be performed relatively quickly by traversing the abstract syntax tree (AST) or concrete syntax tree (CST) of the program, keeping track of declared identifiers and their scopes. This enables the implementation of features such as semantic highlighting, completions, go to definition, find references, renaming, and basic code actions for refactoring implicit variables, all with minimal latency. As a result, developers can benefit from real-time feedback and interactivity, enhancing their overall productivity and development experience.

The performance evaluation conducted in this thesis provides strong evidence to support the efficiency of scope checking for LSP features. The custom CST scope checker implemented as part of this research consistently outperformed the Agda scope checker, and type checker in terms of execution time. On average, the custom scope checker took less than 10 milliseconds to process a file, falling well within the range of instantaneous feedback as perceived by users. This demonstrates that scope checking can indeed provide the fast and responsive IDE features necessary for a smooth development experience.

However, it is important to acknowledge the limitations of scope checking compared to full type checking. Scope checking alone may not catch certain type-related errors or incon-

sistencies that would be detected by a complete type checking process. Additionally, the lack of complete type information can limit the range of LSP features that can be implemented solely based on scope checking. For example, features like type-aware code completion or type-driven refactorings may be more challenging to implement without access to the full type information.

Despite these limitations, the benefits of using scope checking for LSP features in Agda are significant. By providing fast and responsive IDE support, scope checking can greatly improve the developer experience and make Agda more accessible to a wider audience. It can also serve as a complementary approach to full type checking, offering immediate feedback for certain features while more comprehensive type checking is performed asynchronously in the background when the user is not actively editing the file.

To mitigate the limitations of scope checking, potential strategies can be explored. One approach is to investigate incremental type checking techniques, which allow for faster feedback by reusing previously computed type information and only rechecking the parts of the code that have changed. This could help bridge the gap between the responsiveness of scope checking and the completeness of full type checking. Another possibility is to adopt a hybrid approach that combines scope checking and type checking. In this scenario, scope checking can be used to provide immediate feedback for certain features, while type checking is performed only to catch type-related errors and enable more advanced LSP features. This hybrid approach can strike a balance between responsiveness and completeness, offering developers the benefits of both scope checking and type checking.

7.2 Comparison with Existing Agda Development Tools

When comparing the LSP implementation developed in this thesis with existing Agda development tools, it is important to consider both the advantages and limitations of the LSP approach.

One of the main advantages of the LSP implementation is its editor-agnostic nature. By adhering to the standardized Language Server Protocol, the LSP implementation can be integrated with a wide range of editors and IDEs, providing a consistent development experience across different environments. This is in contrast to Agda's Emacs mode, which is tightly coupled with the Emacs editor and may not be easily accessible to developers who prefer other editors and are not accostumed to typical Emacs workflows and key bindings. The LSP approach democratizes Agda development, making it possible for developers to use their preferred tools while still benefiting from advanced IDE features.

The feature comparison conducted in this thesis highlights the strengths and limitations of the LSP implementation compared to Agda's Emacs mode. While the LSP implementation provides a solid foundation for editor integration, code navigation, code assistance, and refactoring, it currently lacks some of the advanced interactive editing features that are available in the Emacs mode. Features like interactive case splitting, hole filling, and proof search are deeply integrated into the Emacs environment and leverage Agda's interactive mode for a seamless development experience. These features are critical for the interactive and exploratory nature of dependently typed programming, and their absence in the current LSP implementation is a notable limitation.

However, it is important to recognize that the LSP implementation developed in this thesis is an initial prototype and serves as a proof of concept for using scope checking as a foundation for LSP features in Agda. The current limitations can be addressed through future work and improvements. Integrating the LSP implementation with the official Agda scope checker, enhancing the range of supported LSP features, and exploring ways to incorporate interactive editing capabilities are all potential avenues for further development.

7.3 The Role of Emacs in Dependently Typed Language Development

The evolution of Emacs has been closely intertwined with the development and adoption of dependently typed languages. In the early years, when languages like Coq (1989), Agda (2007), Idris (2008), and Lean (2013) emerged, the Language Server Protocol (LSP) had not yet been widely adopted. Emacs, with its robust customization capabilities and Lisp-based configuration, became a natural framework for researchers and practitioners working with these complex languages.

The popularity of Emacs within the functional programming community and its strong ecosystem of language-specific modes and tools further solidified its position as the de facto editor for dependently typed languages. Emacs provided a flexible and extensible environment for developing and interacting with the sophisticated type systems and proof assistants that are characteristic of these languages.

However, the landscape has started to shift with the growing prevalence of the Language Server Protocol. Newer iterations of dependently typed languages, such as Idris 2 (2020) and Lean 4 (2021), have embraced LSP integration more extensively. This transition reflects the broader trend towards standardized protocols for communication between editors and language servers. And it is also time for Agda to capitalize on this trend by adopting LSP, thereby enhancing its accessibility and usability across a wider range of editing environments.

7.4 Future Work and Improvements

While the LSP implementation developed in this thesis demonstrates the feasibility and benefits of using scope checking for Agda development, there are several areas for future research and improvement.

One of the primary areas for future work is the implementation of additional LSP features to further enhance the development experience. The current prototype focuses on a core set of features such as semantic highlighting, go to definition, find references, and basic code actions. However, there is significant potential to expand the range of supported features. Implementing more advanced code actions, such as refactoring support for common Agda-specific patterns, could greatly improve the productivity of Agda developers. Integrating code lenses to provide contextual information and actions directly within the editor could also enhance the discoverability and usability of Agda's powerful features. Additionally, supporting features like formatting and workspace-level analysis would provide a more comprehensive LSP implementation and align with the expectations of modern IDE users.

Another critical area for future work is addressing the limitations of the current implementation. One of the key steps in this direction would be to integrate the LSP implementation with the official Agda scope checker, rather than relying on a custom scope checking solution. This integration would ensure compatibility with the latest Agda version and allow the LSP implementation to benefit from any improvements made to the official scope checker. Moreover, fixing the limitations of the Agda parser and scope checker, such as handling error recovery and providing more robust parsing, would greatly enhance the reliability and usability of the LSP implementation. These improvements would require close collaboration with the Agda development team and the Agda community to ensure a seamless integration and a high-quality user experience.

Performance and responsiveness are crucial factors in the success of an LSP implementation, and there is room for further optimization in the current prototype. Investigating incremental type checking techniques could be a valuable direction for future research. Incremental type checking would allow for faster feedback by reusing previously computed

type information and only rechecking the parts of the code that have changed. This approach could help bridge the gap between the responsiveness of scope checking and the completeness of full type checking. However, implementing incremental type checking in Agda would be a significant undertaking and would require close collaboration with the Agda development team.

Engaging with the Agda community and seeking feedback from Agda users is another essential aspect of future work. The LSP implementation developed in this thesis is a research prototype, and its long-term success and adoption depend on its ability to meet the needs and expectations of the Agda community. Contributing the LSP implementation to the official Agda project, or collaborating with the maintainers of existing Agda development tools, could help ensure its long-term sustainability and integration with the broader Agda ecosystem. By actively seeking user feedback and incorporating suggestions, the LSP implementation can be refined and extended to better serve the Agda community. This engagement could take the form of user surveys, workshops, or community discussions to gather insights and prioritize future development efforts.

Chapter 8

Conclusion

In this thesis, we have explored the potential of the Language Server Protocol (LSP) to enhance the development experience for dependently typed languages, focusing on the Agda programming language. Our research demonstrates that scope checking can serve as a solid foundation for implementing efficient LSP features in Agda, offering a promising approach to improve the tooling and overall development experience for dependently typed languages.

The findings of this study demonstrate the effectiveness of scope checking in providing fast and responsive IDE features, such as semantic highlighting, go-to-definition, find references, and basic code actions for refactoring implicit variables. By decoupling these features from the computationally expensive process of full type checking, we can deliver real-time feedback and interactivity, which are essential for a smooth and productive development workflow.

Although the scope checking approach has limitations, such as not supporting certain LSP features, compared to full type checking, the benefits in terms of performance and responsiveness are substantial. The custom scope checker developed as part of this research consistently outperformed the Agda type checker in terms of execution time, making it well-suited for providing real-time feedback in an IDE setting.

This research makes a valuable contribution to the growing body of knowledge on language server protocols and their application to dependently typed programming languages. By demonstrating the feasibility and advantages of using scope checking for LSP features, this thesis paves the way for further enhancements in the development experience of Agda and other similar languages. The insights gained from this research can guide future efforts to improve tooling support and foster the adoption of dependently typed languages in both academic and industrial contexts.

As the adoption of dependently typed languages continues to grow, the importance of providing intuitive, responsive, and feature-rich development tools cannot be overstated. By leveraging the power of LSP and scope checking, we can create more accessible and productive environments for dependently typed programming, enabling researchers and developers to focus on their core tasks and unlock the full potential of these expressive and powerful languages.

Bibliography

Amann, Sven et al. (2016). "A study of visual studio usage in practice". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1. IEEE, pp. 124–134.

Arapakis, Ioannis, Xiao Bai, and B Barla Cambazoglu (2014). "Impact of response latency on user behavior in web search". In: *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pp. 103–112.

Brady, Edwin (2021). "Idris 2: Quantitative type theory in practice". In: *arXiv preprint arXiv:2104.00480*.

Collins, Michael and Brian Roark (2004). "Incremental parsing with the perceptron algorithm". In: *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pp. 111–118.

CoqIDE (n.d.). https://coq.inria.fr/doc/v8.12/refman/practical-tools/coqide.html. Accessed: 2024-06-29.

Coquand, Catarina, Makoto Takeyama, and Dan Synek (2006). "An Emacs interface for type directed support constructing proofs and programs". In: *European Joint Conferences on Theory and Practice of Software, ENTCS*. Vol. 2.

Danielsson, Nils Anders and Ulf Norell (2011). "Parsing mixfix operators". In: *Implementation and Application of Functional Languages: 20th International Symposium, IFL 2008, Hatfield, UK, September 10-12, 2008. Revised Selected Papers 20*. Springer, pp. 80–99.

Faithfull, Alexander et al. (2018). "Coqoon: An IDE for interactive proof development in Coq". In: *International Journal on Software Tools for Technology Transfer* 20, pp. 125–137.

Idris Community (2024a). *Idris2 Language Server*. https://github.com/idris-community/idris2-lsp. Accessed: 2024-05-15.

— (2024b). *idris2-lsp: Language Server for Idris2*. Accessed: 2024-05-17. URL: https://github.com/idris-community/idris2-lsp.

JetBrains (2024). *Language Server Protocol*. https://plugins.jetbrains.com/docs/intellij/language-server-protocol.html. Accessed: 2024-05-22.

LeanProver (2024). *Lean Language Server*. https://github.com/leanprover/lean-client-js/tree/master/lean-language-server. Accessed: 2024-05-15.

MacroMates Ltd. (2024). *Language Grammars*. https://macromates.com/manual/en/language_grammars. Accessed: 2024-06-09.

McCabe, Thomas J (1976). "A complexity measure". In: *IEEE Transactions on software Engineering* 4, pp. 308–320.

Mehnert, Hannes and David Christiansen (2014). "Tool demonstration: An IDE for programming and proving in Idris". In: *Proceedings of Vienna Summer of Logic, VSL* 14.2.

Meyerovich, Leo A and Ariel S Rabkin (2013). "Empirical analysis of programming language adoption". In: *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pp. 1–18.

Microsoft (2023). *Language Server Protocol*. https://microsoft.github.io/language-server-protocol/. Version 3.17. Accessed: 2024-04-26.

Moura, Leonardo de and Sebastian Ullrich (2021). "The lean 4 theorem prover and programming language". In: *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer, pp. 625–635.

Murphy, Gail C, Mik Kersten, and Leah Findlater (2006). "How are Java software developers using the Eclipse IDE?" In: *IEEE software* 23.4, pp. 76–83.

Nawrocki, Wojciech, Edward W Ayers, and Gabriel Ebner (2023). "An extensible user interface for Lean 4". In: *14th International Conference on Interactive Theorem Proving (ITP 2023)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Pacak, André, Sebastian Erdweg, and Tamás Szabó (2020). "A systematic approach to deriving incremental type checkers". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA, pp. 1–28.

Pit-Claudel, Clement F, Pierre Courtieu, and Clément Pit-Claudel (2016). "Company-Coq: Taking Proof General one step closer to a real IDE". In.

Proof General (n.d.). https://proofgeneral.github.io/. Accessed: 2024-06-29.

Rask, Jonas Kjær et al. (2021). "The specification language server protocol: A proposal for standardised LSP extensions". In: *arXiv preprint arXiv:2108.02961*.

Ringer, Talia et al. (2019). "QED at large: A survey of engineering of formally verified software". In: *Foundations and Trends® in Programming Languages* 5.2-3, pp. 102–281.

Shrestha, Prabir (2024). *vim-lsp*. https://github.com/prabirshrestha/vim-lsp. Accessed: 2024-05-22.

Wagner, Tim A and Susan L Graham (1998). "Efficient and flexible incremental parsing". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 20.5, pp. 980–1013.

Zayour, Iyad and Hassan Hajjdiab (2013). "How much integrated development environments (ides) improve productivity?" In: *J. Softw.* 8.10, pp. 2425–2431.

Zwaan, Aron, Hendrik van Antwerpen, and Eelco Visser (2022). "Incremental type-checking for free: Using scope graphs to derive incremental type-checkers". In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2, pp. 424–448.