Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

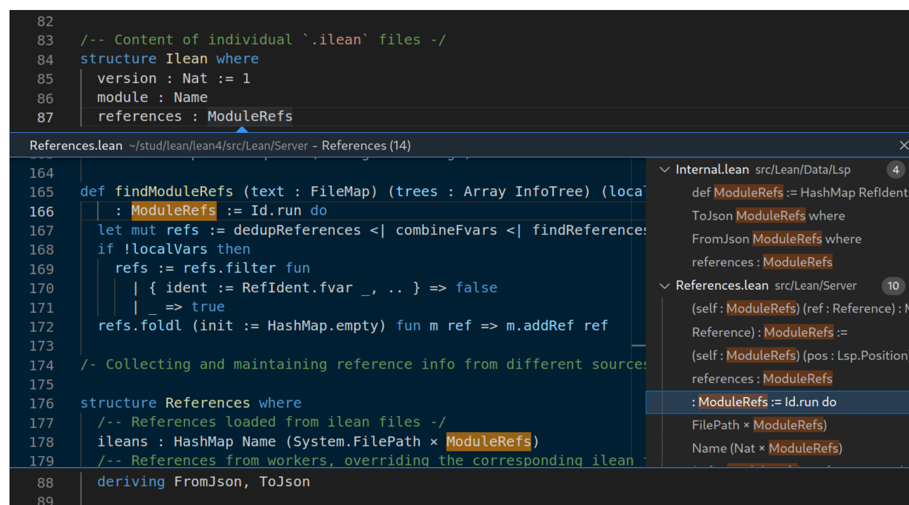
Lehrstuhl Prof. Dr.-Ing. Snelting

Locating and presenting lexical references in a theorem prover

Bachelorarbeit von

Joscha A. Mennicken

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuender Mitarbeiter:	M. Sc. Sebastian Ullrich

Abgabedatum:	5. April 2022

Abstract

Being able to find other references to a symbol is a very useful tool, both when exploring or navigating a code base and when planning and implementing refactorings. However, theorem provers like Lean 4 are usuaully structured around individual files and can't easily answer project-wide queries.

In this thesis, I implement an infrastructure for project-wide symbol information. I design `ilean` files containing symbol information and modify the Lean 4 compiler to output them. I also modify the Lean 4 language server to load and reload `ilean` files as well as incorporate symbol information from files opened and modified by the user. I then use this infrastructure to implement finding references and workspace symbol search as well as to improve the existing go-to-definition implementation.

This implementation has been merged into the Lean 4 project and can now be used via the VSCode and Emacs plugins. Measurements show that it has no significant impact on the compiler or language server performance.

Wenn man Quellcode liest oder durchsucht oder wenn man Refactorings vorbereitet oder durchführt, ist es sehr nützlich, die Referenzen eines Symbols finden zu können. Die Struktur von Theorembeweisern wie Lean 4 basiert jedoch meistens auf individuellen Dateien und ist nicht für projektweite Anfragen geeignet.

In dieser Arbeit implementiere ich eine Infrastruktur für projektweite Symbolinformationen. Dazu entwerfe ich `ilean`-Dateien, die Symbolinformationen enthalten, und modifizieren den Lean-4-Compiler, sodass er `ilean`-Dateien erzeugen kann. Zusätzlich modifiziere ich den Language Server, damit dieser aus `ilean`-Dateien geladene Symbolinformationen sowie Symbolinformationen aus geoffneten und modifizierten Dateien zusammenstellen kann. Diese Infrastruktur verwende ich dann, um Symbol-Referenzen zu finden, projektweite Symbolsuche durchzuführen und die existierende Implementierung von go-to-definition zu verbessern.

Die Implementierung wurde in Lean 4 übernommen und kann jetzt durch das VSCode-Plugin und das Emacs-Plugin verwendet werden. Messungen ergeben, dass die Performanz des Compilers und Language Servers nicht signifikant beeinträchtigt sind.

Contents

1 Introduction **7**

2 Background **9**
2.1 Lean 4 . . . 9
2.1.1 The language . . . 9
2.1.2 The compiler . . . 9
2.2 The Language Server Protocol (LSP) . . . 10
2.3 The Lean 4 language server . . . 11
2.4 Related work . . . 12
2.4.1 The Language Server Index Format (LSIF) . . . 12
2.4.2 Haskell's `hie` files . . . 12

3 Design and Implementation **15**
3.1 Initial design . . . 15
3.1.1 Extracting symbol information . . . 15
3.1.2 Watchdog or worker . . . 17
3.1.3 Watchdog data structures . . . 18
3.1.4 The `ilean` file format . . . 19
3.1.5 The `ilean-bundle` files . . . 20
3.2 Revised design . . . 20
3.2.1 Getting rid of bundles . . . 21
3.2.2 Incremental updates . . . 22
3.3 Further applications of the new infrastructure . . . 22

4 Evaluation **25**
4.1 The size of `ilean` files . . . 25
4.2 Generating and loading `ilean` files . . . 26
4.3 The `textDocument/references` request . . . 27

5 Conclusion **31**

1 Introduction

When working on programming projects, a lot of time is spent reading and navigating the code. Thus, development environments usually include various means of code navigation, for example searching for symbols, going to a symbol's definition or implementation, as well as finding all references of a symbol.

Finding a symbol's references is useful in various situations. When working with unknown code, it can be used to find out how definitions are commonly used. It can be used to check whether all of a symbol's usages are valid, or if the symbol has any usages at all. It is also useful when editing and refactoring code. For example, when replacing a function with a more general function, the original function's references can be updated one-by-one until no more references remain and the original function can be deleted.

Even though it is useful, the Lean 4 language server — used by the Lean 4 VSCode, Emacs, and Neovim plugins — does not support finding references. The server's architecture treats each opened `lean` file individually, with the file only having access to its imports. It has no overview of the entire project. Such an overview is, however, necessary to find all references of a symbol.

In this thesis, I implement an infrastructure to keep track of all symbol definitions and usages in a Lean 4 project. This infrastructure is used to implement finding references as well as to implement or improve a few other LSP requests.

In chapter 2, I will explain the technologies used. Chapter 3 describes the design and implementation process. It describes an initial design, then a revised design addressing issues encountered with the initial design. It also describes applications of the implemented infrastructure. The performance of the final implementation is then measured in chapter 4. Finally, chapter 5 interprets the measurements and lists possible further applications for the implemented infrastructure.

2 Background

2.1 Lean 4

The Lean 4 project [1] is both a general purpose programming language and a theorem prover implemented in that language. It is an open-source project[1] developed mainly by Leonardo de Moura at Microsoft Research and Sebastian Ullrich at KIT, but with contributions from many others.

2.1.1 The language

The Lean 4 programming language is a functional, dependently typed language that can be compiled via C code to binary executables. Similar to Haskell, functions are pure by default and data structures are immutable. An example function in a style similar to Haskell programs can be seen in listing 2.1. Side effects are managed via monads and monad transformers. Lean 4 also has type classes and an extensive do notation [2] that includes early returns, loops, mutable variables and automatic monad lifting. An example for do notation can be seen in listing 2.2. In combination, these features allow writing code in a functional as well as an imperative style.

Lean 4 also has facilities for metaprogramming [3]. Using syntax definitions and macros, the language can be extended at compile-time. This is used to implement many parts of the Lean 4 syntax such as infix notation or the match expression. When proving theorems with Lean 4, this is also used as part of automatic proof searching.

In order to provide a good development experience, Lean 4 plugins for the VSCode, Emacs, and Neovim editors exist. These are based on the Language Server Protocol described in section 2.2 and the Lean 4 language server described in section 2.3, as well as a widget system to interactively display the state of proofs (similar to Lean 3's ProofWidgets [4]).

2.1.2 The compiler

The Lean 4 compiler works one `lean` file at a time.

It first opens the `lean` file and parses as well as *elaborates* its contents. Parsing occurs in chunks that depend on the syntax being parsed which may include custom syntax defined earlier. After being parsed, a chunk is then elaborated. Elaboration is the process of interpreting the parsed syntax, undoing syntactic sugar, executing

[1] https://github.com/leanprover/lean4

```
structure SourceFile where
  name : String
  lines : Nat

def totalLines : List SourceFile → Nat
  | List.nil => 0
  | List.cons { lines, .. } ns => lines + totalLines ns
```

Listing 2.1: Calculating the total amount of lines of a list of source files in an explicit, functional style

```
def totalLines (files : List SourceFile) : Nat := Id.run do
  let mut total := 0
  for file in files do
    total := total + file.lines
  return total
```

Listing 2.2: Redefining the `totalLines` function from listing 2.1 in an imperative style using do notation

macros, type checking, collecting definitions, and executing statements such as `#eval` or `#reduce`.

As elaboration may execute arbitrary code in the form of macros or statements, it may take arbitrarily long. Since theorem proving relies on these capabilities to automate proof finding, loading files containing large proofs can take on the order of minutes, not just seconds. During elaboration, further syntax or macros can be defined. These are then available in the rest of the current file as well as in files importing the current file. This makes the order of imports and the interleaving of parser and elaborator important.

When the compiler has finished loading a file, it can then produce an `olean` and a `c` file. The `olean` file contains a dumped in-memory representation of the file's processed contents. This representation can be loaded and saved with little overhead. When a `lean` file imports another `lean` file, the compiler loads the other file's `olean` file before elaboration begins. When using Lean 4 as a programming language, the `c` files produced this way can be compiled and linked with a C compiler like gcc to produce a binary. When using Lean 4 as a theorem prover, no binary is produced as proofs are verified via type checking. The `c` files can still be used to precompile tactics (which are macros), resulting in shared libraries.

2.2 The Language Server Protocol (LSP)

The Language Server Protocol [5] is a protocol designed for communication between text editors and programming language specific language servers. It was developed

by Microsoft for the VSCode editor but is now used by other editors as well. The protocol tries to solve the problem that every language needs a custom plugin for every editor it wants to support. Since editor plugins work differently from editor to editor, this would mean duplicated work.

The protocol defines communication between a text editor (the client) and a language server. The server can provide language-specific features such as syntax highlighting, information on hover, code completion, go-to-definition, find-references searching for symbols by name, renaming symbols and modules, or reformatting code.

Interaction between a client and a server begins when the user opens a file or project. The editor figures out the programming language and the corresponding language server. The protocol has a concept of workspaces so the editor only needs to start one server process per project. It then starts a new server process and attaches to its stdin and stdout for communication. Messages in the protocol consist of HTTP-like headers followed by a JSON-RPC message as the body. Initially, the client and server negotiate which features to use. This ensures that client and server don't need to implement all parts of the protocol. Then the client notifies the server of changes, requests information or tells the server to perform actions.

2.3 The Lean 4 language server

The Lean 4 language server design is influenced by a few restrictions. Elaborating and processing a file is CPU intensive and may take seconds or even multiple minutes. During elaboration, arbitrary code may be executed. Unloading and reloading of `olean` files would be difficult to implement as they are loaded via memory mapping. This leads to a design where opened source files are handled individually. Instead of using the contents of imported files, which would require the server to load them and all transitive imports, their `olean` files from the last compilation are used.

The server is split up into two main parts, the watchdog and the file workers. The watchdog is the process that is launched by and communicates directly with the client. It spawns one worker for each file opened by the client. The worker then loads the `olean` files of the imports and elaborates the file. When a file is changed, the worker re-elaborates the file from that point on. This means that the worker knows the current state of its own file as well as its imported files' states from the last compilation. Communication between the watchdog and the workers happens using a subset of the Language Server Protocol with custom extensions. The watchdog forwards most file-based LSP requests directly to the worker responsible for the file, which then computes a reply that is forwarded by the watchdog to the client.

As the workers are in separate processes from the watchdog, the CPU intensive processing of their file does not affect the watchdog or other workers, keeping other files responsive in the editor. When a file's imports are changed or the imported `olean` files should be reloaded after a compilation, the watchdog can just restart the worker. The watchdog can do the same if a worker hangs or crashes with no need

to restart the entire server. A disadvantage of this isolation is that cross-file LSP requests such as finding references or searching for a symbol in the entire workspace can't be implemented as they depend on the contents of unopened files for which no workers exist.

2.4 Related work

2.4.1 The Language Server Index Format (LSIF)

The Language Server Index Format [6] is a way to represent and store LSP information about the files in a workspace. While the LSP includes requests that are interactive (e.g. completion, searching) or that modify files (e.g. renaming symbols, formatting), LSIF only describes the contents of files. This includes information related to the file itself, such as folding ranges and locations of links, as well as information related to specific ranges, for example definitions, references, and hover text.

The format is based on a directed graph. The graph is represented as a stream of nodes, vertices and events, represented in JSON. Vertices represent files, ranges in a file, LSP responses, and *monikers*. Different kinds of edges describe the relationship between vertices. Monikers are strings that can be used to identify symbols across LSIF dumps.

LSIF is a flexible format for static dumps of entire workspaces. However, the Lean compiler and language server operate mostly on a single-file basis. In order for LSIF to be used, it would need to be modified slightly, or each file would need to be treated as a separate workspace. Incremental updates from the workers to the watchdog as described in section 3.2.2 would also require some modifications in the format or its interpretation. The format itself is simple to emit but not as simple to interpret due to its flexibility. Because of the complexity of the format and the changes necessary to adopt it for use with the Lean 4 compiler and language server, a simpler format described in section 3.1.4 was developed instead.

2.4.2 Haskell's `hie` files

Starting from version 8.8, the Glasgow Haskell Compiler (GHC) is able to generate `hie` files [7, 8]. They contain a lot of information about their corresponding source file that GHC knows during compilation but which would otherwise be lost once compilation finishes. The files include information such as the types of each subexpression, information on identifiers, and even the source code itself. As they contain a lot of information, there are many possible applications for `hie` files, for example generating LSIF files[2] and static analysis[3].

Once hie files for a project have been generated, hiedb[4] can be used to load multiple

[2] https://github.com/mpickering/hie-lsif
[3] https://github.com/kowainik/stan
[4] https://github.com/wz1000/hiedb

hie files into an SQLite database for fast indexing and querying. `hie` files and hiedb are used by the Haskell Language Server (HLS)[5]. During operation, HLS generates its own `hie` files and loads them into a database using hiedb. This lets HLS persist information across restarts and respond to queries immediately after starting even though it is still loading the project. The database is also used for queries like finding references.

`ilean` files and their use by the Lean 4 language server have similarities to `hie` files. Both can be generated by the compiler and are used by their respective language servers to provide fast queries across entire projects. However, `ilean` files were created specifically for the task of providing reference information and intentionally exclude other information to keep file size and memory usage small. While HLS loads and type checks opened projects and generates its own `hie` files, the Lean language server relies on the compiler's `ilean` files for unopened `lean` files because loading and type checking an entire large proof-heavy project would take too long.

[5] https://github.com/haskell/haskell-language-server

3 Design and Implementation

Finding all references of a symbol in a project requires analyzing the entire project. As discussed in section 2.3, the language server works on files separately and uses the `olean` files from the last compilation to fill in the gaps. However, `olean` files don't contain information about all of a file's symbols including their position, only about definitions and their positions. In addition, file workers only know about the file's dependencies, but not the files depending on their file. One can see how LSP requests like finding a symbol's definition are fairly easy to implement even if the definition is in another file, but finding a symbol's usages requires additional infrastructure.

3.1 Initial design

In the initial design, the compiler extracts the locations of symbol definitions and usages during compilation. For each `lean` file, it saves this information in an `ilean` file placed next to the file's `olean` file. Then, the contents of all these `ilean` files are collected and written into one `ilean-bundle` file. The `ilean-bundle` file only contains the information of existing `lean` files, not old `ilean` files that haven't been cleaned up yet. In the revised design described in section 3.2, `ilean-bundle` files were removed again.

When the language server is started, the watchdog loads all `ilean-bundle` files in the root directories of the current `olean` search path. It also registers with the LSP client to receive notifications whenever an `ilean-bundle` file is created, changed, or deleted. When it receives such a notification and the `ilean-bundle` file is in a valid location, it loads, reloads, or unloads the bundle. The workers send the watchdog their file's symbol information whenever they finish elaboration after their file was changed. Finally, the watchdog responds to incoming find-reference requests using its current symbol information. It does not initiate communication with the workers for this.

3.1.1 Extracting symbol information

Symbol information must be extracted in two places: During compilation and in the LSP worker. Luckily, they both use the same machinery and output `InfoTree`s. An `InfoTree` is a tree-based data structure containing information from the elaboration process, for example the names and types of variables and their location in the source file. During compilation, the collection of `InfoTree`s must be enabled. However, this was measured in section 4.2 to have no significant runtime impact.

Finding identifiers in `InfoTrees` is straightforward: Look through the deepest nodes of every branch (e.g. the leaf nodes) of every `InfoTree`. If it is an identifier in an expression, the tree will include its full name and whether it is a definition or not. If it is a struct field in a constructor, the tree will include its full name.

There are two kinds of identifiers in expressions, global (also called const) and local (also called fvar). Global identifiers can be accessed from other definitions later in the same file and through imports. Local identifiers are restricted to scopes inside definitions or expressions and can't be accessed outside those scopes. Global identifiers are unique across files while local identifiers are not. This means that only global identifiers need to be included in `ilean` files. Examples for the different kinds of identifiers can be found in listing 3.1.

```
1  structure Book_G where
2    title_G : String_g
3    pages_G : Nat_g
4
5  def rename_G (book_L : Book_g) (new_title_L : String_g) : Book_g :=
6    let new_book_L := { book_ℓ with title_s := new_title_ℓ }
7    new_book_ℓ
```

Listing 3.1: Examples for different kinds of identifiers. Global identifiers are marked with $_g$ and their definitions with $_G$. Local identifiers are marked with $_\ell$ and their definitions with $_L$. Struct fields in constructors are marked with $_s$.

A special case where this approach needs to be augmented are parameters in method signatures. Method parameters have two different local identifiers, one inside the method signature (the signature identifier) and one inside the method body (the body identifier). An example for this can be seen in listing 3.2. In terms of references, a parameter's identifiers should not be distinguished. For this, `InfoTree` generation was modified so that the definition of a parameter's signature identifier exactly overlaps the definition of its body modifier. When identifiers are collected, local identifiers with overlapping definitions are treated as the same identifier. In listing 3.2, the identifiers a and b are combined, as well as c and d.

```
1  def mkTuple (t_AB : Type) (v_CD : t_a) : Type × t_a :=
2    (t_b, v_d)
```

Listing 3.2: Example for how method signatures and method bodies use different identifiers for the same parameter. Different identifiers are marked with different letters. Definitions are marked with uppercase letters while usages are marked with lowercase letters. Only local identifiers are marked.

3.1.2 Watchdog or worker

After implementing the extraction of symbol information from `InfoTrees`, I modified the worker to respond to `references` LSP requests. This initial approach was limited to the contents of the worker's file. In order to extend the approach to consider entire projects, parts of the implementation would need to be distributed between the watchdog and the workers. There were three main options.

As described in section 2.3, the workers themselves respond to most LSP requests regarding their file. For cross-file references, more information is required. The first idea was for a worker to request reference information from other workers before responding. This way, the existing infrastructure for registering new request handlers as well as locating symbols at a given source position could be used. However, worker-to-worker communication has no existing infrastructure and would be difficult to implement robustly. For example, a worker may be killed or crash before it has a chance to respond. The overhead of communication may also lead to decreased responsiveness. In addition, it is not clear how information for closed files should be accessed. Each worker could load all `ilean` files, but this is unnecessary duplication of resources. Alternatively, the watchdog could load `ilean` files and respond to requests by workers. This would lead to even more communication complexity and overhead.

The second idea was that the watchdog loads and reloads `ilean` files and responds to requests. When responding, it queries the worker responsible for the request's file for the symbol at the request's position. It then queries all workers for references and uses the `ilean` information for all closed files. This approach solves the problem of where the symbol information should be stored. It also utilizes the existing worker infrastructure for finding symbols and references. Similar to the first approach, this also comes with the disadvantage of complex communication. A request for references would require the watchdog to send a request to each worker and wait for the responses before responding itself. This two-way communication is again difficult to implement robustly and carries with it some amount of overhead for each request.

The third idea — which I ended up implementing — is for the watchdog to contain all information necessary to respond to requests for references. The workers send the watchdog status updates when they finish re-elaborating after their file has been changed. This requires only simple one-way communication. The communication doesn't occur for each request the watchdog receives, meaning that its overhead is not as important. One possible issue is that the watchdog's information may become out-of-date when editing a file. During editing, the worker may not get to fully elaborate the file and send updates to watchdog before the next change causes it to re-elaborate. This issue is solved later via incremental updates, described in section 3.2.2. The watchdog also becomes more complex. Stability and performance issues in the watchdog affect the entire language server, while worker issues stay confined to the worker.

3.1.3 Watchdog data structures

The data structures used by the initial design to hold symbol information in the watchdog can be seen in listing 3.3. Here, the term "reference" means a symbol's definition and its usages.

```
1  inductive RefIdent where -- An identifier for a reference is either
2    | const : Name → RefIdent -- a global identifier or
3    | fvar : FVarId → RefIdent -- a local identifier
4
5  structure RefInfo where -- Information about a reference
6    definition : Option Lsp.Range
7    usages : Array Lsp.Range
8
9  -- All references of a single module (i. e. source file)
10 def ModuleRefs := HashMap RefIdent RefInfo
11
12 -- All modules contained in an ilean-bundle file
13 def Bundle := HashMap Name ModuleRefs -- The keys are module names
14
15 structure References where
16   bundles : HashMap System.FilePath Bundle
17   overlays : HashMap Name (Nat × ModuleRefs) -- The keys are module names
```

Listing 3.3: Data structures for representing symbol information in the initial design. `Name` and `FVarId` are the types used to identify global and local identifiers respectively. `Name` is also used to identify modules. `Lsp.Range` represents a range of characters in a source file, according to the LSP standard. `Nat` is a natural number.

The watchdog stores a `References` value containing the information from the `ilean-bundle` files as well as the information from all currently open files (called `overlays`). These two are kept in separate `HashMaps` to make loading, reloading and unloading of bundles and the opening and closing of files straightforward. In `References.bundles`, there is one entry for each `ilean-bundle` file. In `References.overlays`, there is one entry for each open file. For querying, a `References` object can be converted to a `Bundle` by first combining all bundles into a single bundle and then overwriting `ModuleRefs` objects with their corresponding overlays. If multiple bundles include the same module, one of the conflicting `ModuleRefs` objects is chosen arbitrarily. They are all assumed to come from the same `ilean` file because Lean 4 does not allow defining the same module multiple times.

For open files, the LSP protocol establishes a version number that strictly increases after each change to the file. The workers include this number when sending their file's symbol information to the watchdog. The watchdog saves this number and ignores any update that doesn't increase the version number. This is to avoid

inconsistencies if a worker sends updates out of order.

When receiving a request for references from the client, the watchdog executes two steps: First, it needs to find the symbol for which references were requested. The request itself only includes a file and a position within that file. To find the symbol, the watchdog searches through all symbols in the file's `ModuleRefs` object, obtaining a `RefIdent` or aborting if no symbol is found at the position. Individual files are expected to be on the order of tens to hundreds of lines long, so a linear search is sufficient. Second, the watchdog needs to find all references to the symbol it just found. For this, it looks up the symbol's `RefInfo` in every `ModuleRefs` object. The `RefInfo.definition` field is included (if present) if the request specifies to include declarations, or excluded otherwise. To avoid replying with source locations inside moved or deleted files, only `ModuleRefs` objects for which a corresponding source file exists are included in this second search.

3.1.4 The `ilean` file format

`ilean` files are a straightforward serialization of a `ModuleRefs` object into JSON. An example `lean` file and corresponding `ilean` file can be found in listing 3.4 and listing 3.5 respectively. They include a version number for future changes of the format, as well as the name of the module they have been created from. Positions in the source file are represented as a 4-element list of the form `[start line, start column, end line, end column]` instead of objects in order to keep the files small. `ilean` files only include global identifiers as local identifiers can't be referenced from other files.

```
1  def main : IO Unit :=
2    IO.println "Hello, world!"
```

Listing 3.4: An example lean file `Main.lean` in the module root. The corresponding `ilean` file can be found in listing 3.5.

```
1  {
2    "version": 1,
3    "module": "Main",
4    "references": {
5      "c:IO": { "usages": [ [0,11,0,13] ], "definition": null }
6      "c:IO.println": { "usages": [ [1,2,1,12] ], "definition": null },
7      "c:main": { "usages": [], "definition": [0,4,0,8] },
8      "c:Unit": { "usages": [ [0,14,0,18] ], "definition": null },
9    }
10 }
```

Listing 3.5: The pretty-printed `ilean` file for listing 3.4.

The serialized format of `ModuleRefs` objects is also used for the worker-to-watchdog updates described in section 3.1.2 and section 3.2.2. Here, the `ModuleRefs` objects sent to the watchdog include local identifiers in addition to global ones. A prefix for identifier names is used to distinguish between global (prefix `c:`) or local (prefix `f:`) identifiers.

I chose the format of JSON in text files for multiple reasons. For one, it is easy to use since the Language Server Protocol uses JSON and so JSON support was already implemented. This includes automatic deriving of serialisation and deserialisation implementations based on a data structure's declaration. It also allows me to re-use the same serialisation and deserialisation for worker-to-watchdog communication. As JSON is a text-based format, it is easy to inspect and debug manually. Finally, it is easy to decode and use from other languages.

3.1.5 The `ilean-bundle` files

In addition to `ilean` files, the initial design uses `ilean-bundle` files generated by the build system. An `ilean-bundle` file is a JSON file that can contain the contents of multiple `ilean` files. Lean 4 itself uses a cmake-based build system while Lean 4 projects often use Lake. Lake is a build system for Lean 4 projects and is itself written in Lean 4. Neither of these build systems cleans up its artefacts on every compilation. When a `lean` file is moved or deleted, its corresponding `ilean` file stays around. If the language server loaded and used all `ilean` files, it could lead to conflicts in common editing scenarios. For example, if a file is renamed, the same symbols will be defined in the old and new `ilean` file. Duplicate definitions can also occur if a definition is moved from one file to another.

To solve this problem, the build system generates `ilean-bundle` files after compiling its `lean` files. It includes only the `ilean` files it just produced in the bundle. The language server then loads the `ilean-bundle` files instead of the individual `ilean` files, meaning it has a consistent view of the results of the last compilation. To avoid the issue of outdated `ilean-bundle` files, the build system only generates a fixed number of such files and updates them on every rebuild. A bundle is created by calling a new binary with a list of `ilean` files as argument.

Two options were considered for the structure of `ilean-bundle` files. They could either contain a list of existing `lean` or `ilean` files, or they could contain the contents of the `ilean` files directly. The second option was chosen as it simplified loading the files. In the revised design, `ilean-bundle` files were removed again. See section 3.2.1 for more information.

3.2 Revised design

The revised design tries to address a few issues found during or after implementation of the initial design. It gets rid of `ilean-bundle` files. Instead, the server ignores all `ilean` files for which no corresponding `lean` file exists. This change allows the worker

data structures to be simplified as well. More LSP requests are implemented using the `ilean` infrastructure. In order to keep them as responsive as before, incremental `ilean` updates from workers are implemented. Finally, handling of overlapping symbols at the request positions is fixed.

While the initial design combined local identifiers whose definitions overlapped, it did not consider the fact that global identifier definitions might also overlap. This occurs in `structure` definitions. The structure's name and its constructor function are both defined in the same place. When searching for a structure's references via its definition, either the references of its name or of its constructor function were shown, but not both. In the revised design, references for all definitions overlapping the request position are shown.

3.2.1 Getting rid of bundles

I created `ilean-bundle` files to prevent outdated `ilean` files from being loaded. Aside from that, they could also have been used to solve further issues. If the load time of `ilean-bundle` files had been too long, they could have been structured similarly to `olean` files, allowing them to be loaded via memory mapping. During creation of `ilean-bundle` files, preprocessing like merging the symbol information or deduplicating ranges could have been implemented. When loading `ilean-bundle` files instead of `ilean` files, there are also less files and directories to watch for updates, which could improve the language server's performance when recompiling. However, I encountered none of these issues during development.

On the other hand, `ilean-bundle` files add complexity to the build process. A separate binary is called to create the `ilean-bundle` files, duplicating the `ilean` files' contents in the process. The watchdog already needs to find each module's `lean` file for the response to the request for references. This makes `ilean-bundle` files redundant as the watchdog can simply ignore `ilean` files without a `lean` file.

Loading `ilean-bundle` files only behaves differently from loading `ilean` files directly in very specific cases, for example when deleting a file, then compiling the project and then creating a file with the same name again. Recompiling the project is sufficient to bring the language server's symbol information back into a consistent state. Because `ilean-bundle` files add complexity but no benefit, they were removed.

Listing 3.6 shows the watchdog's new representation of symbol information after the removal of `ilean-bundle` files. `References.bundles` has been replaced with `References.ileans`, with one fewer level of `HashMap` nesting. Next to the `ModuleRefs` objects loaded from `ilean` files, the `ilean` file's path is stored to make unloading easier. Aside from the bug fix described in section 3.2, the implementation of the request for references has not changed. When querying `Requests`, `Requests.ileans` and `Requests.overlays` are still merged into a single `HashMap` from module names to `ModuleRefs` objects.

```
structure References where
  -- The Name keys in these hash maps are module names
  ileans : HashMap Name (System.FilePath × ModuleRefs)
  overlays : HashMap Name (Nat × ModuleRefs)
```

Listing 3.6: Data structures for representing symbol information in the revised design. The definition of `ModuleRefs` is the same as in listing 3.3.

3.2.2 Incremental updates

When a worker responds to an LSP request like `textDocument/definition` or `textDocument/hover`, it doesn't require the file to be evaluated fully. Instead, it waits until enough of the file has been elaborated and then replies immediately. For example, the file only needs to be elaborated up to the request position for a `textDocument/definition` request. In contrast, the initial implementation required the worker to process the entire file and update the watchdog before a request would see the new state.

Incremental updates as implemented in the revised design sit in-between those two options. After a file is changed, the worker sends the information for all unchanged chunks along with the new file version number to the watchdog, replacing the watchdog's earlier (now outdated) information about the file. Whenever it has processed a new chunk of the input file, the worker immediately sends that information to the watchdog as well. When it finishes processing the file, the worker sends the entire file's information to the watchdog again, but this time in a single packet, and the watchdog overwrites its old information with this new information. This is done to avoid inconsistencies that may arise from reassembling partial file information. The watchdog answers requests with only the information it has at the time of the request. It does *not* wait for all relevant chunks to be processed by worker first.

3.3 Further applications of the new infrastructure

Once I implemented the `textDocument/references` request, I used the symbol information infrastructure to implement and augment other requests.

Workspace symbol search via the `workspace/symbol` request lets the user search for a symbol in the current workspace. Since the watchdog has the names of all symbols from the current project as well as any dependencies with `ilean` files in the `olean` search path, this request is easy to implement[1]. Symbols containing the characters of the query in order (but not necessarily consecutively) are returned as the search result.

The `textDocument/definition` request was adapted to search for the definition in the watchdog's symbol information and only pass on the request to the responsible

[1] https://github.com/leanprover/lean4/pull/964

worker if the search was unsuccessful[2]. Previously, it would be forwarded to the watchdog immediately, which would then look up the location of the symbol's definition in the imported `olean` file. If the file containing the definition had been changed since the last compilation, the `olean` file's location information was out-of-date and editors would show the wrong location. Because the watchdog now inspects its symbol information first, the correct location is returned if the file containing the definition is already open.

The `textDocument/documentHighlight` request was implemented by Lars König using symbol information inside the worker[3]. This request lets editors highlight all occurrences of the symbol under the cursor in the current file.

[2] https://github.com/leanprover/lean4/pull/979

[3] https://github.com/leanprover/lean4/pull/969

4 Evaluation

In this chapter, the performance and behaviour of the revised implementation is inspected. The language server being tested is compiled from commit `d2cc5b4a`[1] in the Lean 4 repository. The language server is being used on code from the same commit.

4.1 The size of `ilean` files

This section investigates the size of `ilean` files compared to the size of their corresponding `lean` files. For this, the size of 499 `lean` files and their `ilean` files are graphed in figure 4.1. These are all the `lean` files of the Lean 4 project excluding `Leanc.lean`, `Leanpkg.lean` and the lake submodule.

Using a logarithmic scale, the size relationship appears mostly linear. The `ilean` files tend to be a bit larger than their `lean` files. There is a group of `ilean` files which appear to stay small independently of the size of their `lean` files. There are also a few outliers whose `ilean` files are a lot smaller than the `lean` files, but not small enough to end up in the previously mentioned group.

The linearity is expected because `lean` files usually consist mostly of global definitions that use other global definitions in their bodies. If `lean` files deviate from this structure, the corresponding `ilean` file can shrink. For example, `lean` files with long comments lead to smaller `ilean` files. Similar to comments, imports don't count as symbols either. The same can happen if a `lean` file defines macros or syntaxes as those definitions don't show up in the `InfoTree` in the same way as normal definitions. Yet another way to shrink `ilean` files is to use mostly local symbols like function arguments or `let` or `where` bindings. These effects lead to the outliers in the graph.

The group of `ilean` files with empty references, marked as hollow circles in the graph, exists due to the same effects that produce the other outliers. The Lean 4 project contains quite a few files that exist just to import other files. Sometimes, these importing files also contain a few macro definitions. Even though these `ilean` files contain no reference information, they are not entirely empty and their size varies. This is because every `ilean` file contains a file format version number and the module name of the `lean` file. As the module names vary, the `ilean` sizes vary as well.

[1] https://github.com/leanprover/lean4/commit/d2cc5b4a8325f3da0ceec551769920b2eab4ed70

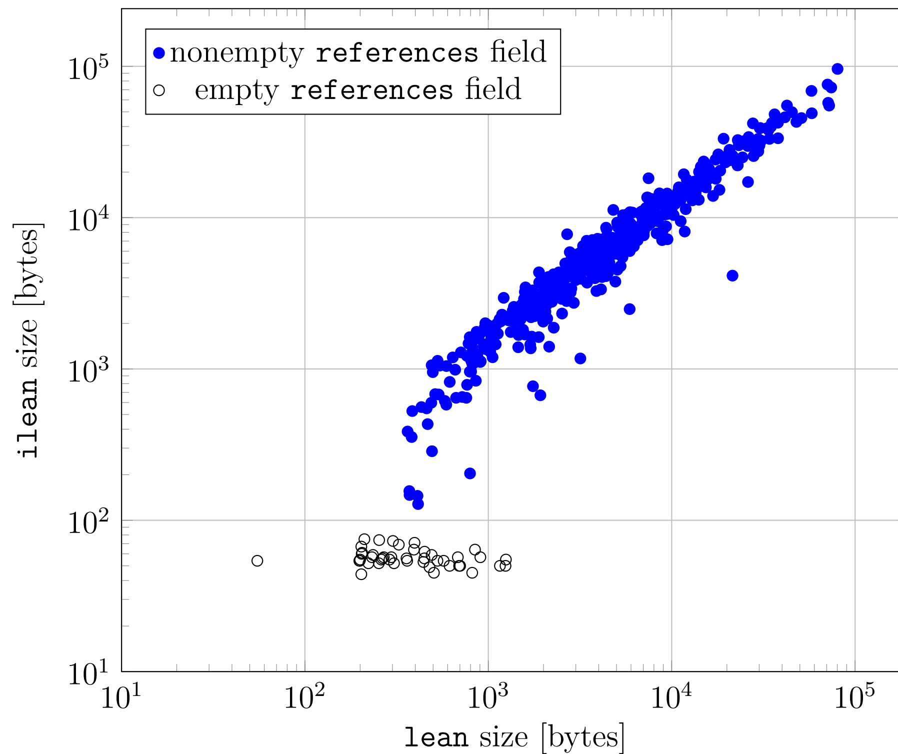


Figure 4.1: Size of `lean` files and corresponding `ilean` files. If the `references` field in the `ilean` file has at least one reference, a solid circle is used. Otherwise, an empty circle is used.

4.2 Generating and loading `ilean` files

In order to generate `ilean` files, `InfoTrees` need to be captured during compilation. This is normally disabled and enabling it has some overhead. In a pull request[2] to the Lean 4 repo, Sebastian Ullrich determined that the performance overhead of enabling it is not too large.

When the language server is initially started, it loads all `ilean` files it can find on the `olean` search path. When the LSP client notifies it of `ilean` file changes, it loads, reloads or unloads the affected files. The Lean 4 project has 545 `ilean` files which the server loads in approximately 300 ms. During compilation, `ilean` files are reloaded as soon as the LSP client detects they have changed. Because the initial `ilean` loading is only performed once and because compilation takes a lot longer than the 300 ms a full `ilean` reload takes, the overhead of loading `ilean` files is not noticeable in practice.

Most individual `ilean` files take less than 1 ms to load, with the longest load time being about 6.5 ms. Figure 4.2 shows the load times of all `ilean` files, averaged over several full `ilean` reloads. The load time appears to be roughly proportional to the file size, with a nonzero overhead per file. There is a single reference-less `ilean`

[2] https://github.com/leanprover/lean4/pull/834

file with a longer load time than all other reference-less `ilean` files. It is the file `Std/Data.ilean`, which was loaded first on every full `ilean` reload. Even though multiple such reloads were performed consecutively, this `ilean` file consistently took longer to load.

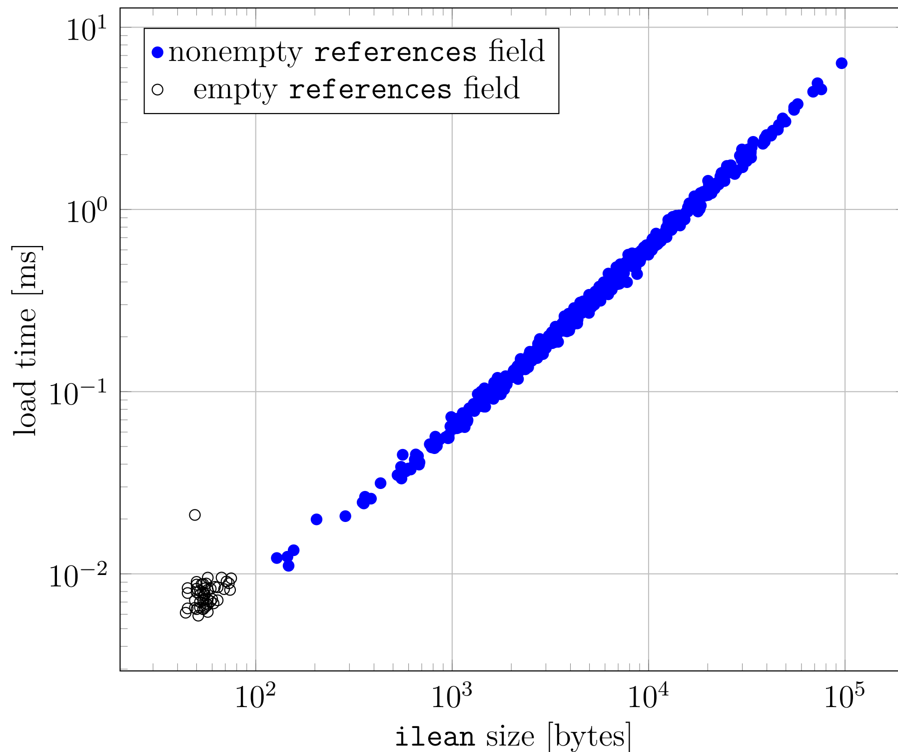


Figure 4.2: Size and load time of `ilean` files. If the `references` field in the `ilean` file has at least one reference, a solid circle is used. Otherwise, an empty circle is used.

The resident set size of the watchdog with all `ilean` files loaded is approximately 75 MiB. When loading no `ilean` files, its resident set size is approximately 47 MiB. In comparison, a single worker might use multiple hundred mebibytes of memory. For example, a worker for `Lean/Server/Watchdog.lean` uses 250-300 MiB of memory and a worker for `Lean.lean` uses 350-400 MiB of memory. Compared to those, the watchdog's overhead is acceptable. Since workers don't store their symbol information after sending it to the watchdog, their memory usage while not elaborating a file is not affected.

4.3 The `textDocument/references` request

This section investigates the performance of the `textDocument/references` request implementation for the 20 most commonly used symbols in the Lean 4 code base. For this, multiple requests were made per symbol. The request location was the

symbol's definition. Only the file containing the definition was open at the time of the request. Times measured were rounded up to the nearest millisecond.

The table in figure 4.3 shows two different times for each symbol. The search time is the time it took the server to find the symbol based on the request position as well as to then find the symbol's references. The total time includes serializing and printing the response. The table also includes the amount of occurrences of each symbol as well as the amount of files it occurs in. The amount of files a symbol is used in does not directly depend on the amount of occurrences. Some symbols like `Bool` and its constructors are used in most files while other symbols like `Lean.Syntax` are confined to a smaller set of files but used similarly frequently.

Symbol	References	Files	Search [ms]	Total [ms]
`Lean.Expr`	3332	170	5	30
`Option.some`	2187	288	8	24
`Array`	2153	238	7	23
`Option.none`	2020	289	8	23
`Lean.Name`	1995	226	7	21
`Nat`	1954	236	7	21
`Pure.pure`	1933	242	7	21
`Bool`	1930	293	8	22
`Bool.false`	1416	253	7	18
`Lean.Syntax`	1341	108	4	14
`Lean.Meta.MetaM`	1290	136	4	14
`Bool.true`	1243	232	7	16
`Option`	1217	246	7	16
`String`	1075	145	5	13
`List`	1057	169	5	13
`Unit`	1055	200	6	14
`Array.size`	1039	150	5	13
`Lean.Syntax.getOp`	757	67	3	9
`Array.push`	661	160	5	10
`Lean.Elab.Term.TermElabM`	627	43	3	7

Figure 4.3: Time required to find references of the 20 most commonly used symbols in the Lean 4 code base. The search time includes resolving the request position and finding the references. The total time includes the search time as well as serializing the response. Both times are rounded up to the nearest millisecond. The table includes how often the symbol is referenced in the source code and in how many files it appears.

A symbol's search time seems to follow the amount of files the symbol is in. The symbol's total occurrences don't seem to have a big influence, meaning the search time is dominated by per-file overhead. During the search, the module of each loaded

`ilean` file containing the requested symbol is resolved, resulting in the path of the `lean` file defining the module. Then, this path is resolved into an absolute path so it can be converted into a file URL. These two steps are the main source of per-file overhead. If they are bypassed by using empty strings as the file URLs, the search time for `Option.some` drops from 8 ms to about 2 ms and the total time from 24 ms to about 11 ms.

The total time follows the amount of references more closely than the amount of files, although in some cases the per-file overhead discussed in the previous paragraph is large enough to make a difference. For example, the total time for `Option` is larger than the total time for `Lean.Syntax` even though the latter has over 100 more references. The reason for this seems to be serialization of the response into JSON, as writing the serialized response to stdout and flushing it only takes a few microseconds.

The serialization overhead depends on the size of the response. The response includes a file URL and a range for each reference, leading to redundant file URLs if there are multiple references in the same file. This leads to responses multiple hundred kibibytes in size. For example, the response to a `textDocument/references` request for `Option.some` is approximately 350 KiB large. The reduction of the total time when replacing all file URLs with empty strings is also explained by the serialization overhead.

5 Conclusion

I have implemented reference and symbol search for the Lean 4 language server. For this, I developed the `ilean` file format and modified the compiler to produce `ilean` files. I also modified the server to load symbol information from `ilean` files and to combine the information with incremental symbol information from the individual file workers. Finally, I implemented the `textDocument/references` and `workspace/symbol` LSP requests and fixed the `textDocument/definition` request to jump to the correct source position in more cases.

As can be seen in chapter 4, the implementation's performance is adequate for interactive use. References requests take tens of milliseconds to complete, which is short enough to not cause noticeable delays. The additional overhead during compilation is also small enough to not be a nuisance. Other requests like the workspace symbol search may take longer than 100 ms with specially crafted queries, but this is still fast enough for a search. The workspace symbol search implementation was modified by others to include fuzzy search functionality between my implementation and this evaluation.

When I was familiarizing myself with the Lean 4 code base, go-to-references would often have been useful. Once the implementation was far enough along for basic reference searching, I began using it immediately. By now, it has been merged into Lean 4's `master` branch[1][2][3] and is being used by others[4].

In the future, the infrastructure introduced by this implementation could also be used to implement even more LSP requests or features. It represents a view of the entire project based on the last compilation for unopened files and the current content of opened files. Previously, no part of the language server had a view of the entire project. Now, parts of it are already used to implement the `definition` and `documentHighlight` requests and other features like renaming symbols, code lenses showing the amount of usages[5] or warnings about unused symbols could also benefit from it. As `ilean` files are JSON and thus easy to parse from most languages, programs for static analysis of projects via their symbol infoormation are also easier to write. Possible examples would be programs to detect unused symbols, to lint identifiers, or to prevent symbols whose namespace doesn't match the file's module name from being defined. External programs operating on `ilean` files could also benefit from including more information in `ilean` files, similar to the applications of

[1] https://github.com/leanprover/lean4/pull/835
[2] https://github.com/leanprover/lean4/pull/925
[3] See also section 3.3
[4] https://github.com/leanprover/vscode-lean4/issues/156
[5] https://github.com/leanprover/lean4/pull/975

hie files described in section 2.4.2.

Bibliography

[1] L. de Moura and S. Ullrich, "The lean 4 theorem prover and programming language," in *Automated Deduction – CADE 28* (A. Platzer and G. Sutcliffe, eds.), (Cham), pp. 625–635, Springer International Publishing, 2021.

[2] "The do notation." https://leanprover.github.io/lean4/doc/do.html. Retrieved: 25 Mar. 2022.

[3] S. Ullrich and L. de Moura, "Beyond notations: Hygienic macro expansion for theorem proving languages," in *Automated Reasoning* (N. Peltier and V. Sofronie-Stokkermans, eds.), (Cham), pp. 167–182, Springer International Publishing, 2020.

[4] E. W. Ayers, M. Jamnik, and W. T. Gowers, "A Graphical User Interface Framework for Formal Verification," in *12th International Conference on Interactive Theorem Proving (ITP 2021)* (L. Cohen and C. Kaliszyk, eds.), vol. 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 4:1–4:16, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021.

[5] "Language Server Protocol Specification - 3.16." https://microsoft.github.io/language-server-protocol/specification.html. Retrieved: 23 Mar. 2022.

[6] "Language Server Index Format Specification - 0.6.0." https://microsoft.github.io/language-server-protocol/specifications/lsif/0.6.0/specification. Retrieved: 23 Mar. 2022.

[7] Z. Duggal, "HIE Files - coming soon to a GHC near you!." https://www.haskell.org/ghc/blog/20190626-HIEFiles.html. Retrieved: 29 Mar. 2022.

[8] "hie files." https://gitlab.haskell.org/ghc/ghc/-/wikis/hie-files. Retrieved: 29 Mar. 2022.

Erklärung

Hiermit erkläre ich, Joscha A. Mennicken, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

______________________________	______________________________
Ort, Datum	Unterschrift