DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of a Linter for Isabelle

Yecine Megdiche

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of a Linter for Isabelle

Entwicklung eines Linters für Isabelle

Author:	Yecine Megdiche
Supervisor:	Prof. Tobias Nipkow
Advisors:	Lukas Stevens Fabian Huch
Submission Date:	15.09.2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2021 Yecine Megdiche

Acknowledgments

First of all, I want to thank Prof. Tobias Nipkow for allowing me to do the thesis at his chair. I would like as well to thank both my supervisors, Fabian Huch and Lukas Stevens, for helping me throughout this thesis. From the discussions before starting the thesis to their last remarks before the submission, they were extremely helpful, patient, and open to discuss whatever ideas and suggestions I had.

Moreover, I am thankful to my family, my parents Mourad and Basma, and my siblings, Zeineb and Mohammed, since without them, nothing I achieved would have been possible. I also want to thank my extended family for their encouragement and support, especially my Grandparents Mohammed and Saloua, and Mohammed and Mouna, for flooding me with prayers and good wishes whenever I was blessed to see them. Special thanks are also due to my aunt Mouna for proof-reading my thesis.

Finally, I would like to thank Ons for her constant love and support and for also being patient while listening to me enthusiastically explain the details of the thesis.

Abstract

While the interactive theorem prover Isabelle can assist with developing intricate formalizations leveraging the power of interactive proofs, not all of them have the same quality. Indeed, some proofs might require a higher maintenance effort or be harder to read and understand. Some of the patterns causing these unwieldy formalizations are identified through the years by the Isabelle community. A prominent resource demonstrating these patterns and how to avoid them is *Gerwin Klein's Style Guide for Isabelle/HOL*. However, as it stands, there is no existing tool to automatically warn users of these pitfalls or suggest better alternatives. We attempt to fill this gap in the Isabelle environment by developing an Isabelle *linter*. The linter offers basic configurability, extensibility, Isabelle/jEdit integration, and a standalone command-line tool. With the help of the 20 implemented checks, it uncovered 252 problems in *Isabelle/HOL*, 28.97 % of which are of high severity, 58.35 % of medium severity, and 14.68 % of low severity. Adding to that, 20 randomly-selected entries from *the Archive of Formal Proofs* are analyzed, which produced 575 lints distributed as follows: 45.39 % of high severity, 44.17 % of medium severity, and 10.43 % of low severity.

Contents

Acknowledgments iii

Abstract iv

1. Introduction 1

2. Related Work 3

3. Preliminaries: Isabelle 5
- 3.1. Isabelle/Scala and Isabelle/ML 5
- 3.2. The PIDE Document model 5
- 3.3. Isabelle Syntax 6
- 3.4. IDE Support 6

4. A Linter for Isabelle 8
- 4.1. Overview and Architecture 8
  - 4.1.1. Parsing the outer syntax 9
- 4.2. Lints 9
  - 4.2.1. Lint abstractions 9
  - 4.2.2. The Lint Store 13
- 4.3. Lint Reporting 14
- 4.4. Lint Configuration 15
  - 4.4.1. Options 16
- 4.5. Integration 16
  - 4.5.1. Isabelle/jEdit Integration 16
  - 4.5.2. The `isabelle lint` Command Line Tool 19

5. Evaluation 22
- 5.1. Approach 22
- 5.2. Results 22
  - 5.2.1. Report summary 22
  - 5.2.2. Performance 23

6. Conclusion **27**
6.1. Future Work 27

A. Lints and Bundles **29**

B. XML and JSON reporting example **33**

List of Figures **35**

List of Tables **36**

Bibliography **37**

1. Introduction

Isabelle makes it possible to formalize and verify mathematical concepts in an expressive and interactive way by constructing machine-verifiable proofs: these can be, for example, executed to ensure that a change in an implementation does not affect the correctness of the underlying algorithm. However, not all proofs are created equal: some may be harder to read, or be more prone to break because of new releases or minor changes in their dependencies. Consider this hypothetical example [7]:

```
apply clarsimp
apply (rule my_rule)
 apply (fastforce simp: foo)
proof safe
  fix new
  assume "something surpising"
  show "unforeseen"
```

The proof starts with iterative tactic applications (the `apply` commands) and then switches to a structured Isar proof, where the goals are explicitly stated. The structured proof is prone to break if the goals generated by the apply-script slightly change, which may occur, for instance, due to improvements in the simplifier. It is also hard to read and understand without running Isabelle. A better alternative might be to rewrite this into a fully structured proof.

Correctly identifying these problematic constructs – especially if they are hidden deep inside a lengthy theory with complicated formalizations – requires time, effort, and an experienced eye that knows the standards to adhere. For these reasons, this manual process does not scale.

In this thesis, we aim to help make Isabelle proofs more future-proof by developing a linter. Linters are static analysis tools that help catch bugs and warn about bad practices. By filling this gap in the Isabelle ecosystem, it can help catch problems that might hinder readability and maintainability as early as possible e.g. interactively while the proof is being written, or at a later point on finished theories. The linter is implemented in *Isabelle/Scala* and can be used with *Isabelle/jEdit* or through a dedicated command line tool, `isabelle lint`. It includes 20 checks, mainly from *Gerwin Klein's Style Guide for Isabelle/HOL* [7, 8].

Outline

First, Chapters 2 and 3 examine the required background through covering linters in programming languages, illustrating how the quality of formalizations is maintained in proof assistants, and introducing key concepts of the Isabelle environment needed throughout the thesis.

After that, Chapter 4 details the implementation of the linter and outlines its architecture and the design choices that shaped its development.

Next, Chapter 5 evaluates the performance of the linter and the quality of some selected theories (Isabelle/HOL and some entries from the Archive of Formal Proofs (AFP)[1] with the help of the developed lints.

Finally, the last Chapter summarizes the work and gives an outlook to motivate further developments of the linter.

[1] `https://www.isa-afp.org/`

2. Related Work

The term *lint* in software originates from the *Lint* program developed by S. C. Johnson in 1977, which checks C programs "for bugs and obscurities" [6]. For example, it performs more involved type-checking that is not part of C-Compilers at the time, due to efficiency reasons. Nowadays most programming languages have linters, like ESLint [3] for JavaScript, HLint [9] for Haskell, and pylint [13] for Python. They provide feedback to users to help them catch bugs early, and learn the best-practices of the respective languages. Common features of these linters include IDE and CI/CD integration, and automatic application of the suggestions generated.

When it comes to Isabelle, Gerwin Klein published a style guide for Isabelle proofs on his blog, separated in two parts [7, 8]. It contains a list of anti-patterns that should be avoided while writing proofs and what to do instead. These bad practices can result in a proof being brittle, hard to read, or difficult to reason about, making it harder to maintain and work with. Although it is a good reference for the best practices and the potential pitfalls of working with Isabelle, it is just text: it cannot be run in a continuous integration pipeline, or be used to check to what extent a theory respects its suggestions. It is up to the user to follow it and ensure that proofs conform to that standard.

To automate some of these checks, the `thylint` GitHub action[1] provides a basic linter for Isabelle. It was developed as part of the seL4 [15] project to guarantee that no pull request contains unwanted commands, like proof-finder commands (e.g. `sledgehammer`) or diagnostic commands (e.g. `print_simpset`). It also offers basic configuration support by allowing users to control which classes of commands to prohibit. The limitation of this tool is that it offers neither a more granular control on what constitutes an illegal command and nor integration with the IDEs used for Isabelle. However, it does its job: it prevents merging contributions with unwanted commands.

The situation is similar for most other proof assistants. Coq [1] includes a development style guide on its GitHub repository [2]. Projects using Coq, like Vericert [4], provide their own guides on what is expected from code within their source. The Agda [12] standard library also includes a style guide highlighting best-practices [3]. It is interesting to note that its description states the need for a linter to automate these

[1] `https://github.com/seL4/ci-actions/tree/master/thylint`

[2] `https://github.com/coq/coq/blob/master/dev/doc/style.txt`

[3] `https://github.com/agda/agda-stdlib/blob/master/notes/style-guide.md`

checks: "It is hoped that at some point a linter will be developed for Agda which will automate most of this.". The `mathlib` library [14] for the Lean [11] proof assistant has a dedicated linter [2], that can be invoked at any point with the `#lint` command.

3. Preliminaries: Isabelle

3.1. Isabelle/Scala and Isabelle/ML

Isabelle is implemented in two main programming languages: *Scala* and *Standard ML*. In the context of Isabelle, these are however referred to as *Isabelle/Scala* and *Isabelle/ML*, since the code style is tied tightly to Isabelle. The *Isabelle/Isar Implementation* manual describes Isabelle/ML as a "certain culture based on Standard ML. Thus it is not a new programming language, but a certain way to use SML at an advanced level within the Isabelle environment."[23].

The two implementation languages play different roles in Isabelle. *The Isabelle System Manual* highlights this difference by comparing them to *mathematics* and *physics*: Isabelle/ML corresponds to implementing the tools for the mathematics of Isabelle (like tactics or proofs), whereas Isabelle/Scala is for physics, i.e., to interface with other tools and systems of the "outside world" like IDEs [21].

3.2. The PIDE Document model

Isabelle/PIDE stands for *Prover IDE*, which is the framework managing the editor, the prover, and other tools around the Isabelle proof assistant [18, 20]. The document model is at the heart of the protocol: it decouples interactive exploration of the sources in the editor from the parallel processing of theories by the prover. Modifying the sources or scrolling down the buffer creates new document *versions*, which prompt the prover to interrupt its processing or process the newly visible content as needed. The prover communicates its execution results through PIDE XML markup over the input sources and through messages. The editor uses this information to provide semantic information about the sources through syntax highlighting, underlines, and other GUI elements. Decoupling rendering and editing from the prover process is possible through document *snapshots*, that represent the state of the document at a particular time point [18].

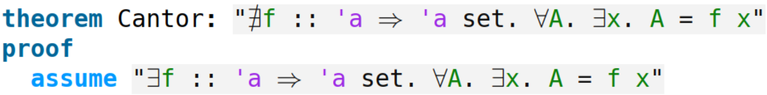
The document model is how Isabelle/Scala and Isabelle/ML are connected. The internal protocol is responsible for communication between the two worlds, with the sources in Isabelle/Scala imitating those in Isabelle/ML: there are many `.ML` files with corresponding `.scala` counterparts to have the same abstraction in both implementation frameworks [17].

3.3. Isabelle Syntax

The Isabelle syntax is split into two parts as described in the *The Isabelle/Isar Reference Manual* [22]:

- The *Outer Syntax*, representing the *theory* language. It covers the proofs, specifications, and outlines of theories.
- The *Inner Syntax*, representing the *term* language. It is used to specify types and logic terms. Inner syntax elements occur as atomic entities in the outer syntax elements.

Figure 3.1 shows the difference between the two syntax categories: the text with a light-grey background corresponds to *inner syntax*, the rest is *outer syntax*.



```
theorem Cantor: "∄f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x"
proof
  assume "∃f :: 'a ⇒ 'a set. ∀A. ∃x. A = f x"
```

Figure 3.1.: Syntax example from HOL/Examples/Cantor.thy

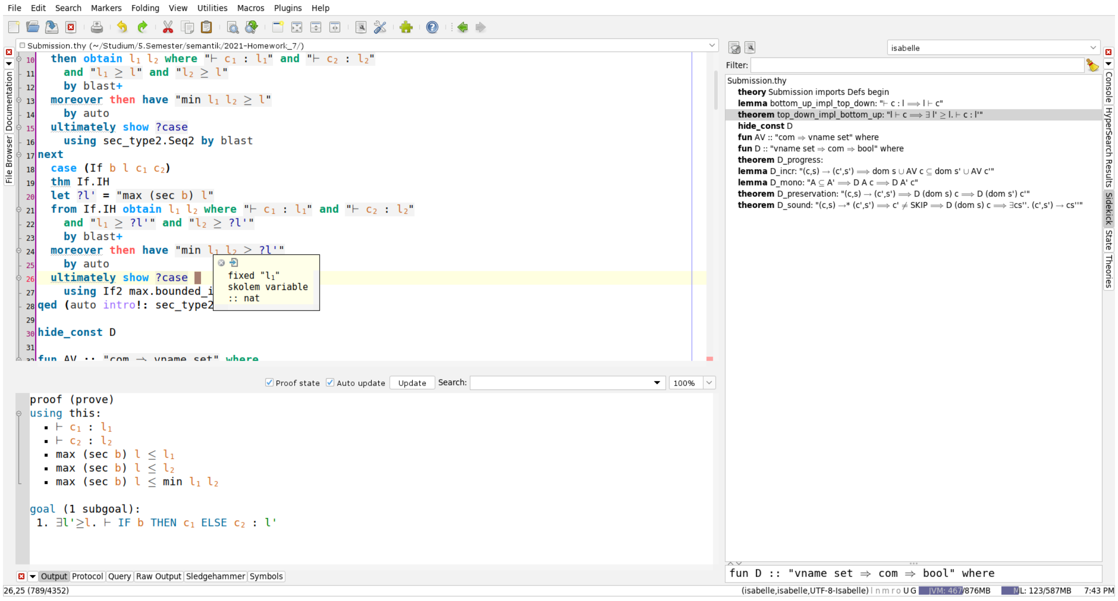
3.4. IDE Support

The primary way to interact with Isabelle is through Isabelle/jEdit [19]. It is the de facto IDE for Isabelle. It consists of the jEdit text editor with the Isabelle plugin, which provides its own panels and functionalities. The main focus is to enable asynchronous and parallel processing of documents. *Dockable* windows are a central part of the jEdit editor providing functionality beyond editing text (like searching or exploring the document structure). They are heavily utilized in the Isabelle plugin to extend the capabilities of the IDE, for example to display the prover messages and command results in the *Output* panel.

Another editor that can be used with Isabelle is VSCode [1] through the *Isabelle/VSCode* extension [20]. It leverages Isabelle/PIDE for *Language Server Protocol*[2] functionality.

[1] `https://code.visualstudio.com/`

[2] `https://microsoft.github.io/language-server-protocol/`



□ Submission.thy (~/Studium/5.Semester/semantik/2021-Homework_7/)

```
  then obtain l₁ l₂ where "⊢ c₁ : l₁" and "⊢ c₂ : l₂"
    and "l₁ ≥ l" and "l₂ ≥ l"
    by blast+
  moreover then have "min l₁ l₂ ≥ l"
    by auto
  ultimately show ?case
    using sec_type2.Seq2 by blast
next
  case (If b l c₁ c₂)
  thm If.IH
  let ?l' = "max (sec b) l"
  from If.IH obtain l₁ l₂ where "⊢ c₁ : l₁" and "⊢ c₂ : l₂"
    and "l₁ ≥ ?l'" and "l₂ ≥ ?l'"
    by blast+
  moreover then have "min l₁ l₂ ≥ ?l'"
    by auto
  ultimately show ?case
    using If2 max.bounded_i
qed (auto intro!: sec_type2

hide_const D

fun AV :: "com ⇒ vname set" where
```

```
fixed "l₁"
skolem variable
:: nat
```

```
proof (prove)
using this:
  ▪ ⊢ c₁ : l₁
  ▪ ⊢ c₂ : l₂
  ▪ max (sec b) l ≤ l₁
  ▪ max (sec b) l ≤ l₂
  ▪ max (sec b) l ≤ min l₁ l₂

goal (1 subgoal):
 1. ∃l'≥l. ⊢ IF b THEN c₁ ELSE c₂ : l'
```

Submission.thy
- **theory** Submission imports Defs begin
- **lemma** bottom_up_impl_top_down: "⊢ c : l ⟹ l ⊢ c"
- **theorem** top_down_impl_bottom_up: "l ⊢ c ⟹ ∃ l' ≥ l. ⊢ c : l'"
- **hide_const** D
- **fun** AV :: "com ⇒ vname set" where
- **fun** D :: "vname set ⇒ com ⇒ bool" where
- **theorem** D_progress:
- **lemma** D_incr: "(c,s) → (c',s') ⟹ dom s ∪ AV c ⊆ dom s' ∪ AV c'"
- **lemma** D_mono: "A ⊆ A' ⟹ D A c ⟹ D A' c"
- **theorem** D_preservation: "(c,s) → (c',s') ⟹ D (dom s) c ⟹ D (dom s') c'"
- **theorem** D_sound: "(c,s) →* (c',s') ⟹ c' ≠ SKIP ⟹ D (dom s) c ⟹ ∃cs''. (c',s') → cs''"

```
fun D :: "vname set ⇒ com ⇒ bool" where
```

26,25 (789/4352)

(isabelle,isabelle,UTF-8-Isabelle) l n m r o U G JVM: 467/876MB ML: 123/587MB 7:43 PM

Figure 3.2.: Isabelle/jEdit in action

The `Isabelle-emacs` project [3] is a similar implementation aiming to bring the same functionalities to emacs.

[3] `https://github.com/m-fleury/isabelle-emacs`

4. A Linter for Isabelle

This chapter is the central part of the thesis: it goes through the implementation of the linter, outlines its architecture, and showcases how it can assist developing better Isabelle theories. As a rolling example, we will examine the *Use by* lint, which suggests replacing

```
lemma: "..."
  apply method
  apply method
  done
```

with the more compact form

```
lemma: "..."
  by method method
```

4.1. Overview and Architecture

The linter is implemented in Isabelle/Scala. It checks the theories based on the outer syntax (e.g. it does not process terms). The main loop of the linter is as follows:

- A document snapshot and a lint configuration (Section 4.4) are supplied to the linter.
- The document snapshot is used to extract the commands, which are then preprocessed to a suitable internal representation by saving their ranges in the source, as well as the ranges of each of the underlying tokens. The linter also tries to *parse* each command into an *abstract syntax tree* (AST) node. (Section 4.1.1).
- Finally, a lint report is created (Section 4.3) based on the lints specified in the configuration.

Interacting with the linter is facilitated through the `Linter_Variable` class. It processes *Isabelle options* to create a suitable configuration for the linter. The variable has a binding to a `Linter_Interface` object which handles communication between the linter and other Isabelle components. The interface caches the results of linting the most recent snapshot of each document, which is a necessary optimization since the same report might be needed multiple times. For example, the Isabelle/jEdit integration of the linter underlines text in the buffer while displaying a detailed description of the checks in a separate panel (see Section 4.5.1 for more details). The results' format is also customizable via the `Linter_Variable`. See Section 4.3 for details on reporting.

This approach results in low coupling between the linter and any interfacing component. For instance, the linter does not require any knowledge about the components using it (the only dependency of the linter is Isabelle). From the perspective of the components using the linter, the convenience of using the Isabelle options for configuration eliminates the need of knowing anything about its internals. The linter consumes the snapshots provided and the result is transformed into the desired format. Using the *variable* design, which splits invoking a component from configuring it, is also common in the Isabelle/Scala code-base (for example, for completion history [19]). Figure 4.1 shows a high-level overview of the interaction with the linter.

4.1.1. Parsing the outer syntax

The grammar parsed by the linter is only a subset of the complete Isabelle grammar. This is justified for two reasons: First, most of the grammar is not relevant for the set of lints implemented, thus the overhead of parsing the complete grammar was spared. Second, Isabelle theories can introduce new syntax elements at runtime that cannot be accounted for by the developed parser. The grammar currently recognized represents the syntax elements related to proof methods. Figure 4.2 shows the railroad diagrams corresponding to that grammar. These are from *The Isabelle/Isar Reference Manual* [22]. The definition of the non-terminals *name*, *args* and *nat* are omitted.

The parsing itself is achieved through *parser combinators* which is a well-known parsing technique in functional programming languages [5, 16]. The implementation builds on top of the `scala-parser-combinators` library [10] and provides Isabelle-specific parsers. For example, `pCommand("apply")` parses the `apply` token in the `apply` command, and `pMethod` parses a proof method into its respective AST node.

4.2. Lints

4.2.1. Lint abstractions

A lint is defined by a name, a severity (either Low, Medium, or High), and a function that takes a list of commands representing the current document snapshot and a report; and returns a new report after potentially adding its results. This general abstraction is

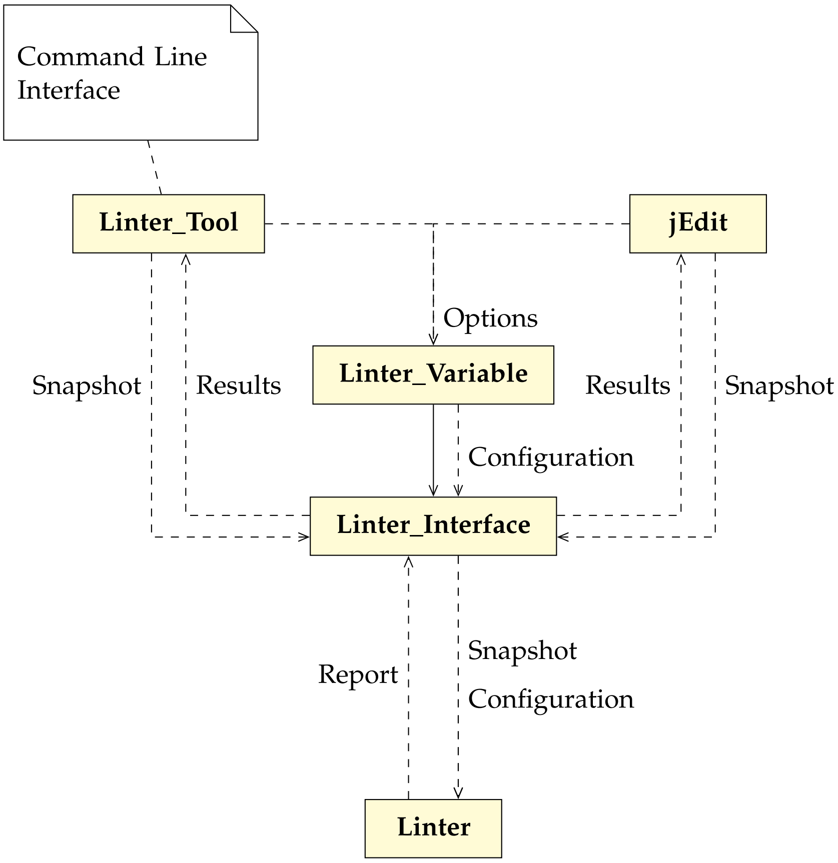


Figure 4.1.: Architecture Overview

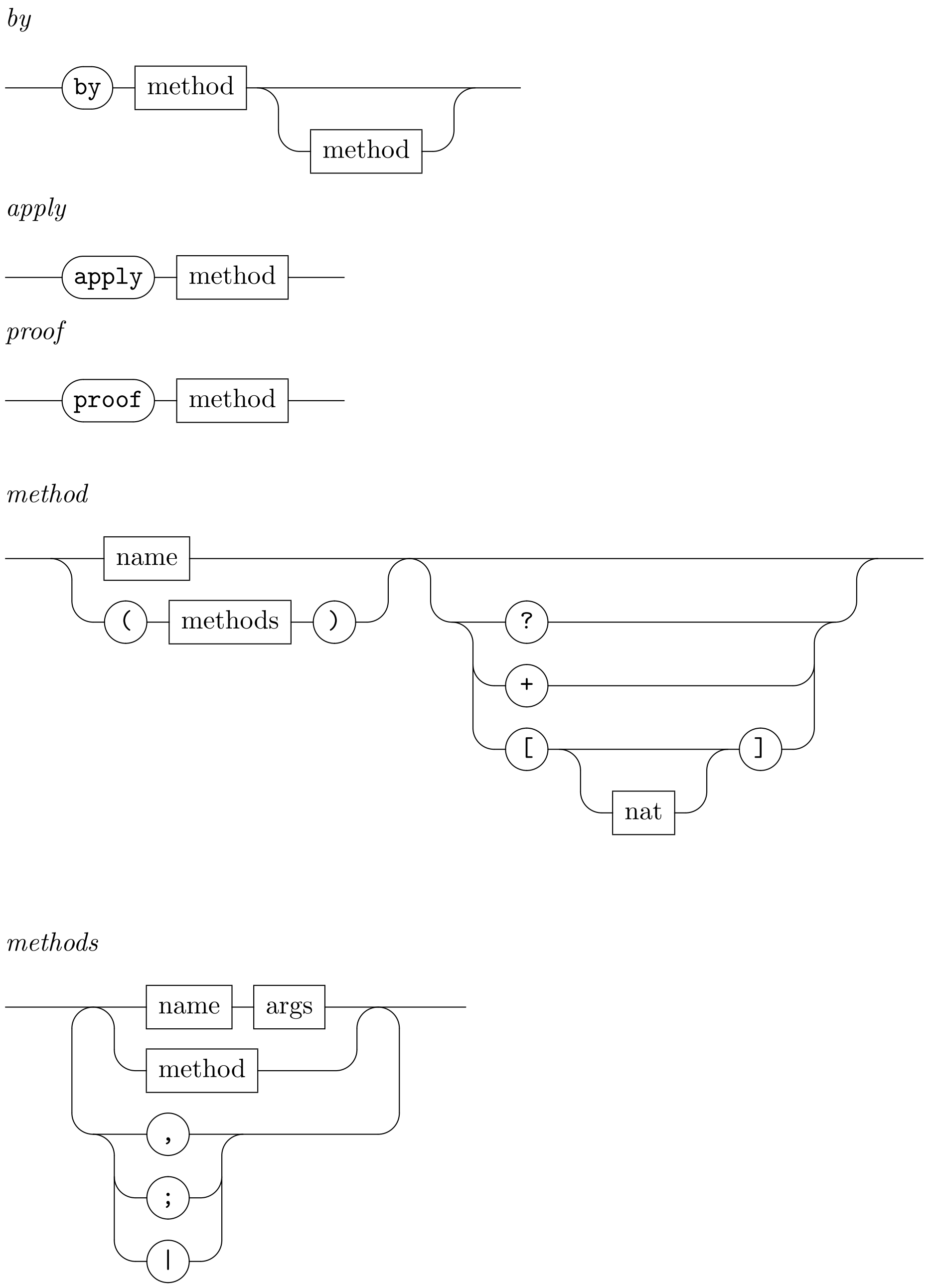


Figure 4.2.: Grammar parsed by the linter

refined further to reduce boilerplate and code duplication and make describing lints more convenient:

- The *proper commands lint* abstraction is used for lints that are concerned with multiple *proper* commands (e.g. no white-spaces or comments). Lints using this must provide an implementation for `lint_proper`, which receives a filtered list of commands. As an example, the *Low Level Apply Chain* lint detects a long chain of single rule applications (like `simp` or `rule`), which could be replaced by automated search methods.
- *Single command lint* is another abstraction used for lints that act on one command. These represent the majority of the lints implemented. This could be used as-is, but it comes with two more refinements:
  - *AST Lint*, which makes use of the parsed AST. It allows the implementing lint to only focus on syntax elements that it is concerned with: if a lint only considers the `proof` commands at the start of Isar-style proofs, then overriding the `lint_isar_proof` function is enough. An example using this abstraction is the *Implicit Rule* lint, which warns about using the `rule` method without explicitly stating the rule used. By overriding `lint_method`, all that is left is to pattern-match the method supplied and check whether it is an implicit rule.
  - *Parser lint*, which is useful since the parsed grammar does not cover all commands available in Isabelle. This offers an abstraction to define lints as parser combinators: they try to parse a lint result based on the command tokens. The construction of the parsers is facilitated through the Isabelle-specific parser combinators used to parse the AST. Next to being flexible, this approach allows for a declarative description of what should be avoided: for example, it can be used to detect when an unnamed lemma has a `simp` or `cong` attribute. The *Global Attribute on Unnamed Lemma* lint implements this check.

To avoid the boilerplate of adding the meta-data like the lint's name or severity, the `add_result` method can used with *proper command lints* and a *reporter* callback with *single command lints*.

The motivating example, *Use by*, aims to express the short apply script in a denser format. For that reason, it is a *proper command lint* with low severity. We could implemented as follows:

```
object Use_By extends Proper_Commands_Lint {

  val name: String = "use_by"
  val severity: Severity.Level = Severity.Low

  def lint_proper(
    commands: List[Parsed_Command], report: Lint_Report
  ): Lint_Report = ...
```

Pattern-matching is used for finding the command sequence "lemma, apply, apply, done": [1]

```
commands match {
  case Parsed_Command("lemma")
      :: (apply1 @ Parsed_Command("apply"))
      :: (apply2 @ Parsed_Command("apply"))
      :: (done @ Parsed_Command("done"))
      :: next => ... // Update the report,
                      // continue checking the remaining commands
  case ... // The rest of the cases
}
```

Thereupon, all that is left is to generate the replacement and update the report, which we continue in section 4.3.

4.2.2. The Lint Store

The `Lint_Store` is like a repository that allows referencing lints by their names. Lints can be registered at runtime, which is useful when the linter is used as a library or Scala code is run interactively in Isabelle theories. This permits externally defined checks to be integrated seamlessly with the linter.

Adding to that, the store introduces the concept of *bundles*. Bundles are groups of lints that can be used together. Depending on the context, the set of lints used can be different: it might be acceptable to have interactive commands like `sledgehammer` while developing proofs in Isabelle/jEdit, but not when trying to submit an entry to the AFP. Bundles can be presets intended to be used on their own (like a preset for interactive proof development) or to group lints that are related (like a bundle to

[1] Other forms can be also rewritten with "by", but they are not discussed in the example. The actual implementation tries to cover all cases.

prohibit interactive commands). As is the case with lints, bundles can also be registered at runtime.

4.3. Lint Reporting

When a lint is triggered, it creates a `Lint_Result` containing all the relevant information: the name of the lint and its severity, a message briefly explaining the lint, the range in the document that is problematic, the list of commands related to the lint as well as an optional `Edit` that specifies a range in the document and with what it should be replaced. These results are accumulated in a `Lint_Report`, which is basically a wrapper around a list of results with some convenience methods, like getting the results for a specific command.

Back to our `use_by` example. To generate the replacement for

```
apply method1
apply method2
done
```

we can use parser combinators to extract the method: parse and ignore the `apply` token followed by a potential white-space, and return the rest of the tokens as a string:

```
  private def removeApply: Parser[String] = (
    (pCommand("apply") ~ pSpace.?) // Parse apply, followed by a
potential white space
      ~> // Ignore what is already parsed
        pAny.* // Accept everything that follows..
        ^^ mkString // .. and turn it to a string
  )

  private def gen_replacement(
    apply_script: List[Parsed_Command]
  ): Option[String] =
    apply_script match {
      case apply1 :: apply2 :: done :: Nil =>
        for {
          method1 <- tryTransform(removeApply, apply1)
          method2 <- tryTransform(removeApply, apply2)
        } yield s"by $method1 $method2"
      case  ... // omitted
    }
```

All that is left is to add the result to the current report using the add_result method:

```
private def report_lint(
  apply_script: List[Parsed_Command], report: Lint_Report
): Lint_Report = {
  val new_report = for {
    replacement <- gen_replacement(apply_script)
  } yield add_result(
    """Use "by" instead of a short apply-script.""",
    list_range(apply_script.map(_.range)),
    Some(Edit(list_range(apply_script map (_.range)), replacement)),
    apply_script,
    report
  )
  new_report.getOrElse(report)
}
```

The described Lint_Result and Lint_Report structures are intended to be used *internally* by the linter. Through Reporters, the reports can be transformed into a suitable format depending on the context. Three reporters are implemented for these purposes:

- XML_Lint_Reporter: returns an XML representation of the report.
- JSON_Reporter: returns a JSON representation of the report.
- Text_Reporter: returns a textual representation of the report.

These reporters are all currently in use: XML is useful within Isabelle as with the Isabelle/jEdit integration; JSON and text are used with the command line interface of the linter (see section 4.5 for more details).

4.4. Lint Configuration

Users should have full control over exactly what lints they want to enable. This is crucial, as the context highly influences which checks are relevant: for example, using the axiomatization command is necessary to create a new logic, but it might result in inconsistencies if carelessly used in formalizations. Configurability is achieved through the Linter_Configuration class. It can be used to enable or disable individual lints, as well as bundles. The selection is based on names, which are subsequently used to fetch the corresponding lints with the help of the Lint_Store.

4.4.1. Options

Isabelle manages persistent settings through *options*. These are stored under `$ISABELLE_HOME_USER/etc/preferences` and managed through Isabelle/Scala. The linter defines a set of configuration options that get applied when interacting with the linter through the `Linter_Variable`:

- `linter` specifies whether the linter is enabled.
- `enabled_bundles` is comma-separated list of the names of the bundles to be enabled.
- `enabled_lints` and `disabled_lints` are additionally two comma-separated lists of the names of lints to be enabled or disabled, respectively.

The last three options are used to generate a corresponding `Lint_Configuration`, by first adding the specified bundles and enabled lints and then removing the disabled lints.

4.5. Integration

4.5.1. Isabelle/jEdit Integration

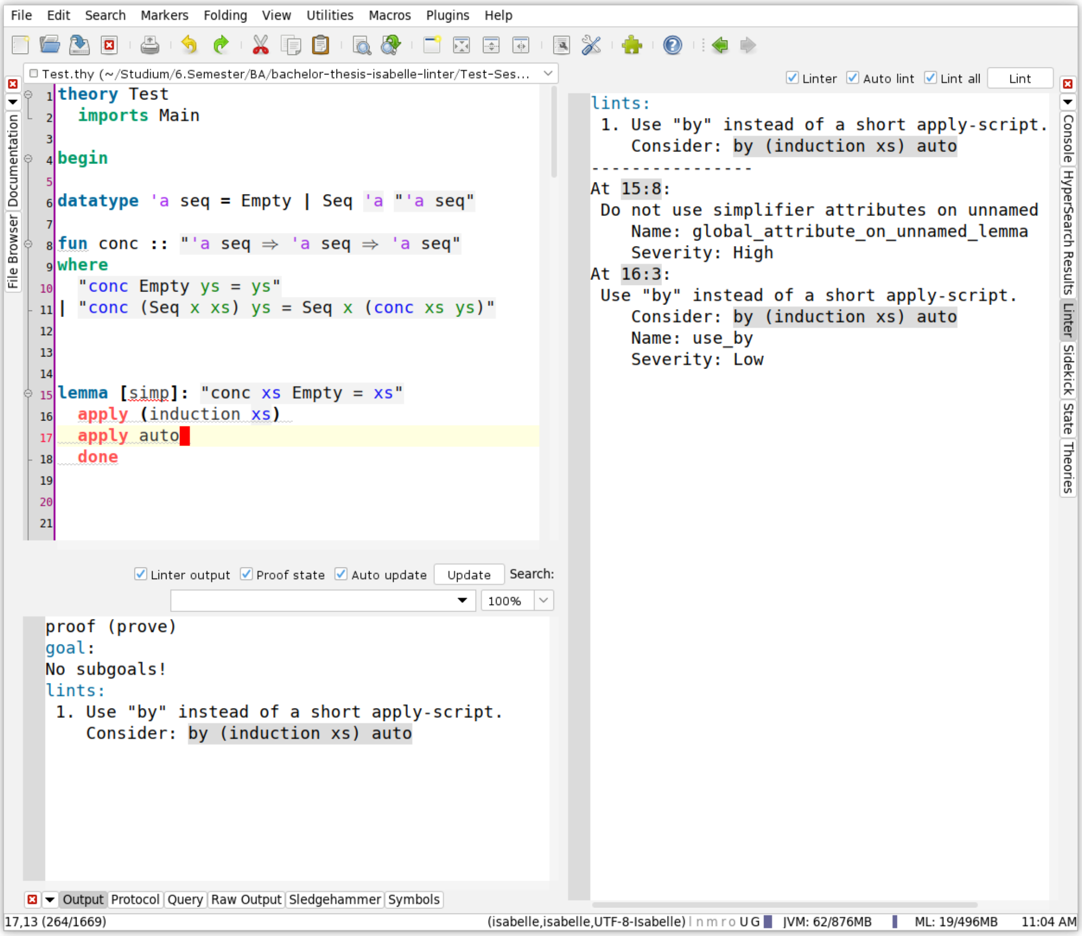
Isabelle/jEdit is the default user-interface and IDE when working with Isabelle, so it was important to integrate the linter with it. The coupling is realized through the PIDE plugin, by providing a binding to a `Linter_Variable` with a `XML_Reporter`. The developed integration provides both feedback on the lints discovered in the active theory, as well as configuration options to customize the behavior of the linter.

Feedback

The results of linting the current theory are available through three main sources (as shown in Figure 4.3):

- *Underlined text* in the main buffer with a color based on the severity of the lint.
- *The output panel* (at the bottom) is central to the interaction with Isabelle/jEdit since it shows the prover messages of the command under the caret. The lints for that command are also appended to the output panel. The displayed lints are sorted in descending order of severity.

- *The linter panel* (at the right) is a new panel introduced specifically for the linter. Like the output panel, it displays lints regarding the current command. Furthermore, it provides a more general overview of the entire theory, with each lint's severity, name, and location included.



☐ Test.thy (~/Studium/6.Semester/BA/bachelor-thesis-isabelle-linter/Test-Ses...

```
theory Test
  imports Main

begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
  "conc Empty ys = ys"
| "conc (Seq x xs) ys = Seq x (conc xs ys)"



lemma [simp]: "conc xs Empty = xs"
  apply (induction xs)
  apply auto
  done
```

☑ Linter ☑ Auto lint ☑ Lint all

```
lints:
 1. Use "by" instead of a short apply-script.
    Consider: by (induction xs) auto
----------------
At 15:8:
 Do not use simplifier attributes on unnamed
    Name: global_attribute_on_unnamed_lemma
    Severity: High
At 16:3:
 Use "by" instead of a short apply-script.
    Consider: by (induction xs) auto
    Name: use_by
    Severity: Low
```

☑ Linter output ☑ Proof state ☑ Auto update

```
proof (prove)
goal:
No subgoals!
lints:
 1. Use "by" instead of a short apply-script.
    Consider: by (induction xs) auto
```

17,13 (264/1669) (isabelle,isabelle,UTF-8-Isabelle) I n m r o U G JVM: 62/876MB ML: 19/496MB 11:04 AM

Figure 4.3.: Isabelle/jEdit with linter integration

Both panels make it possible to automatically apply lint suggestions. The suggestions are displayed in the panels with a grey background (see Figure 4.3). The linter panel also enables navigation to the lint location, by clicking on the reported lint positions (also with a grey background).

To achieve this interactivity, the linter defines two special markups (*lint_edit* and *lint_position*) and adds them to the list of active markups in Isabelle/jEdit. When the user clicks on text marked with these markups, jEdit goes through the defined handlers [2] which perform the actions needed based on the type of markup.

Isabelle users might recognize this feature from using `sledgehammer`: the found proofs are displayed with a grey background, and upon clicking on them, they get inserted into the buffer. This is implemented in the same way as the linter feature: `sledgehammer` wraps the proofs in the *sendback* active markup, and the default active markup handler of Isabelle/jEdit handles inserting the proof.

Configuration options

Adding to the options in subsection 4.4.1, five additional ones are used exclusively for Isabelle/jEdit:

- `lint_all` indicates whether the lints of the whole document should be included in the linter panel
- `lint_output_panel` indicates whether the lints are added to the output panel
- `linter_high_color`, `linter_medium_color` and `linter_low_color` indicate the color of the underline.

The first two options and the ones described in subsection 4.4.1 are available to view and edit under *Plugins / Plugin Options / Isabelle / General / Linter* (together with the rest of the Isabelle options). The color-specific options can be customized under *Plugins / Plugin Options / Isabelle / Rendering*.

The output panel comes with an additional checkbox to control whether the linter output is appended to the prover output. The linter panel has controls to toggle the linter plugin entirely, to indicate whether its output should be updated automatically, to control whether the lints of the whole document should be displayed, and a button to trigger a lint (useful when auto-linting is disabled). These are displayed in Figure 4.3.

Updating results

The linter Isabelle/jEdit integration is implemented in an event-driven approach. A subclass of *Linter_Variable*, called *PIDE_Linter_Variable*, listens to changes in the commands or the global options. When such event is dispatched, it lints the current snapshot in parallel and notifies the listening components when it is done [3] so they can react accordingly.

[2] defined in `src/Tools/jEdit/src/services.xml`

[3] This is achieved by emitting a *Caret Focus* event. Ideally, a separate event for the linter should be added, but that requires more changes to the PIDE plugin. This can be added when a tighter linter integration is required.

4.5.2. The `isabelle lint` Command Line Tool

With the `isabelle lint` tool the linter can be run from the command line. It has the following usage:

```
Usage: isabelle lint [OPTIONS] SESSION

  Options are:
    -b NAME      base logic image (default "Pure")
    -d DIR       include session directory
    -o OPTION    override Isabelle system OPTION (via NAME=VAL or NAME)
    -v           verbose
    -V           verbose (General)
    -r MODE      how to report results (either "text", "json" or "xml",
    default "text")
    -l           list the enabled lints (does not run the linter)

  Lint isabelle theories.
```

The options are as follows:

- Option `-b` provides a base logic image, as used in `isabelle dump` [21].
- Option `-d` allows adding more directories to have access to more sessions.
- Option `-v` increases the verbosity level of the linter.
- Option `-V` increases the general verbosity level.
- Option `-o` enables overriding Isabelle options. This can be primarily used to configure the linter by overriding the relevant options: `enabled_bundles`, `enabled_lints` and `disabled_lints`. The `linter` option is ignored: it does not matter what value it has, the linter will be enabled regardless.
- Option `-l` generates a lint configuration based on the Isabelle options, and prints the results. If this option is set, the linter does not run.
- Option `-r` specifies the reporting mode, which can be `text`, `xml`, or `json`. The `text` mode provides results in a human-readable format. Figure 4.4 shows an example of its usage. On the other hand, both `json` and `xml` options can be used to provide a machine-readable output, which is convenient because the results can be processed further, for example to create warnings as a part of a CI/CD pipeline or to analyze the findings (as done in chapter 5). Example outputs of these two modes can be found int Appendix B.

```
$ isabelle lint -v HOL
Loading 2 sessions ...
Starting session Pure ...
Loading 111 theories ...
...
Processing theory HOL.Inductive ...
At 203:3:                [use_by]
  Use "by" instead of a short apply-script.
  Severity: LOW

apply (erule gfp_upperbound [THEN subsetD])
  apply (erule imageI)
  done

  Suggestion: by (erule gfp_upperbound [THEN subsetD]) (erule imageI)
 ...
```

Figure 4.4.: Output of `isabelle lint` text mode

The tool invokes the linter through a `Linter_Variable`, as is the case with the Isabelle/jEdit integration, in order to handle Isabelle options. The main difference between the two interfaces is that the tool does not cache lint results since they are only needed once. The tool works similarly to the `isabelle dump` command; by processing the PIDE session database on the spot [21]. This is the cause behind the main limitation of `isabelle lint`: it is slow (see Chapter 5). The bottleneck is not the linting but it is rather generating the snapshot. The provided session and all its dependencies need to be processed. Processing `HOL` alone requires substantial memory and time [21]. Option `-b` allows overcoming this problem by providing a base logic image which will be skipped. A possible faster alternative is to rely on the build database to access the document snapshots instead of generating them directly. However, *Isabelle2021* does not provide access to the command spans through the build database, which makes the snapshot unusable for the linter: the whole theory is one unparsed command.

5. Evaluation

In this chapter we evaluate the linter in "real-world" conditions, in order to evaluate its performance. Moreover, the experiments give insight to what extent the best practices that are checked by the linter are applied in various Isabelle theories.

The tests are performed on a machine with a 10 Core 2.4 GHz processor and 46 GB of RAM, running CentOS 8 and Isabelle with the Isabelle2021 February version.

5.1. Approach

The linter is evaluated against two classes of Isabelle theories: the HOL theories, intended to represent theories from official Isabelle libraries, and a random set of 20 sessions from the Archive of Formal Proofs [1]. The bundles of lints employed are respectively the *foundational* and the *afp* bundles.

The command line tool is utilized to perform these experiments. For the AFP sessions, a base logic image is supplied (option `-b`) to each session in order to speed up retrieving the snapshots. The values related to timing are averaged across 10 runs. Crucially, however, these values only consider the time taken to lint the theories, without including the time needed to fetch the snapshots (which is dominant when using the command line tool).

5.2. Results

5.2.1. Report summary

The report summary for linting Isabelle/HOL and the selected AFP sessions can be seen on Table 5.1 and Table 5.2 respectively. The linter detected a total of 252 lints in Isabelle/HOL and 575 lints in the AFP sessions. This is a significant difference, considering the sizes of both sets: Isabelle/HOL consists of 111 theories with around 113 thousand lines of theories, whereas the AFP selection contains 135 theories with

[1] Lp, LTL_Master_Theorem, Constructive_Cryptography_CM, Recursion-Addition, Randomised_Social_Choice, Parity_Game, LTL, Possibilistic_Noninterference, IMP2_Binary_Heap, DataRefinementIBP, Transitive-Closure, Stewart_Apollonius, Minkowskis_Theorem, Stellar_Quorums, Smooth_Manifolds, Category2, VerifyThis2019, No_FTL_observers, BDD, Pairing_Heap

Severity	Name	Number of occurences
High	Unrestricted auto	71
	Global attribute on unnamed lemma	2
Medium	Complex Isar initial method	62
	Implicit rule	55
	Complex method	20
	Lemma-transforming attribute	3
	Apply-Isar switch	2
Low	Use by	37
		Total: 252

Table 5.1.: Lint summary of Isabelle/HOL

around 73 thousand lines. In relative terms, one lint got triggered every 448 lines in Isabelle/HOL versus every 126 lines in the AFP sessions. The distribution of these lints is however comparable: those with high severity were 28.97 % of the lints detected in Isabelle/HOL, which is lower than the 45.39 % in the AFP sessions. Respectively for medium severity lints it is 58.35 % versus 44.17 % and 14.68 % versus 10.43 % for the low severity lints.

5.2.2. Performance

The median time taken to lint a theory is 20.7 milliseconds with a mean of 53.55 milliseconds. This means that the results should be viewed skeptically: the theory sample used has a bias towards smaller theories. In fact, the median length of the theories is 452 lines with a mean of 771.5 lines, but the longest theory, *HOL.List*, has 8199 lines. Another caveat to mention is that the number of lines of a theory is not the only metric that could be applied to quantify the size of a theory. However, it is a simpler and more common property than the number of commands.

Figure 5.1 and Figure 5.2 show the time taken to process a theory, relative to the number of lines it has, and the number of results reported. We could observe two main aspects: First, larger theories (in terms of number of lines) tending to take longer to process, and second, the processing time tending to increase with the number of lints, for theories of similar length.

To get more insight on where time is spent during linting, the BDD session from the AFP is linted with the VisualVM[2] profiler attached. For reference, the session has a total of 11058 lines of theory from which the linter generated 72 suggestions. Data from

[2] https://visualvm.github.io/

Severity	Name	Number of occurences
High	Unrestricted auto	233
	Global attribute on unnamed lemma	27
	Counter-example finder	1
Medium	Complex method	81
	Complex Isar initial method	70
	Apply-Isar switch	68
	Implicit rule	33
	Lemma-transforming attribute	2
Low	Use by	60
		Total: 575

Table 5.2.: Lint summary of the AFP selection

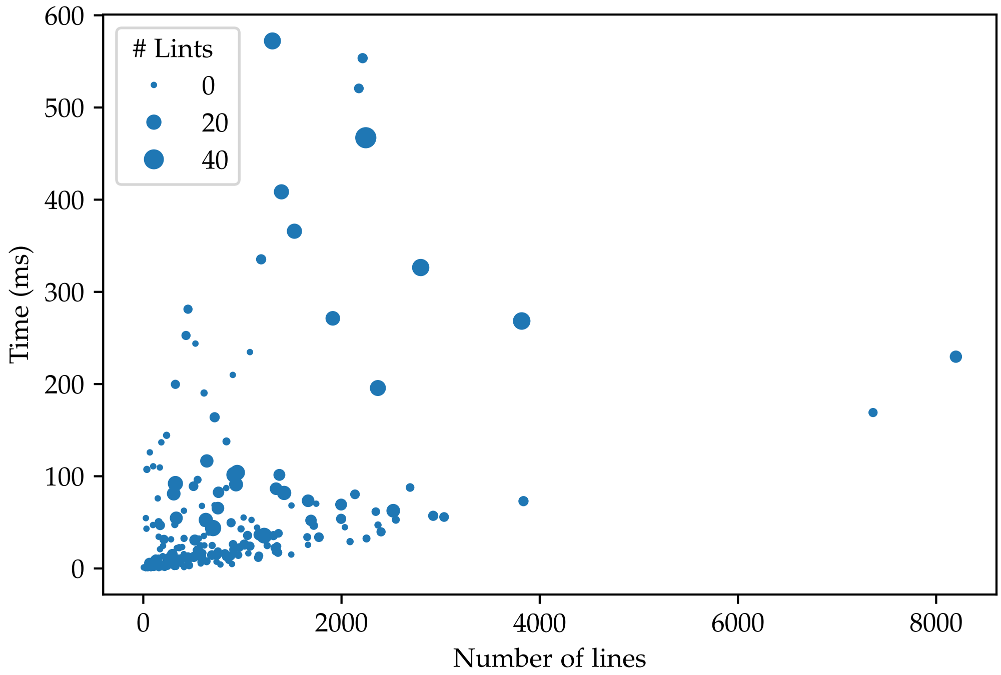


Figure 5.1.: Time taken to lint a theory depending on its length and the number of lints

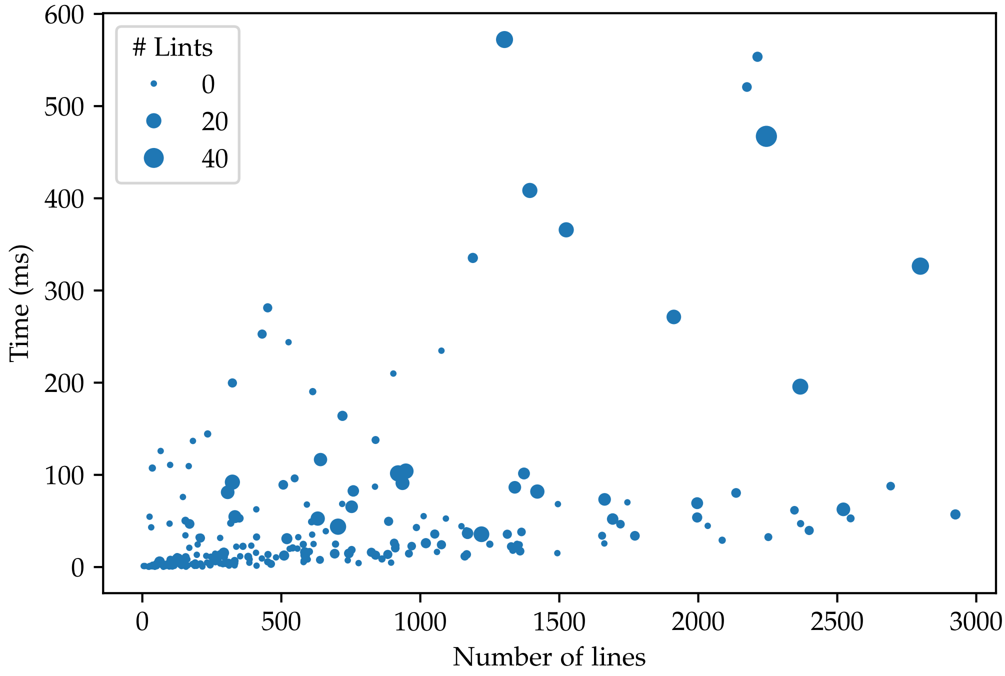


Figure 5.2.: Time taken to lint a theory depending on its length and the number of lints (restricted to theories shorter than 3000 lines)

the profiler suggests that 31.67 % of the time is spent by the linter trying the different checks, and the rest 68.33 % is spent by the reporter converting the results to the required format (JSON in this case). The reporter spent almost all the time converting text offsets to the "line, column" format. Text offsets represent how many characters are there before a certain position, which represents the way position information is communicated within Isabelle. The more usual "line, column" format makes it easier for users to navigate the sources and facilitates integration with other tools like Language Server Protocol clients. Creating reporters that do not convert text offsets to lines and columns could cause a significant speedup in *relative* terms. In absolute terms, for example, it took the linter a total of 1039 milliseconds to process the BDD session, meaning around 710 milliseconds were spent in the reporter. This is not a noticeable difference, especially when considering the time taken to also generate the snapshots: around 2 minutes and 31 seconds are spent in total during the invocation of the tool. The time effectively needed by the linter to process the theories is rather insignificant.

6. Conclusion

The lint support in automatic proof assistants is underwhelming when compared with programming languages. Although guides and standards exist, they could not offer automated feedback to users: it is the responsibility of the proof writers and maintainers to learn about these standards and try to abide by them.

In this thesis, we attempt to fill this gap. We explored and implemented a linter for Isabelle in Isabelle/Scala, that involves 20 checks. In addition, it provides Isabelle/jEdit integration to assist with interactive proof development and a command line tool available through `isabelle lint`. *Extensibility* and *configurability* were two main goals of the design. With the help of the linter, we generated 252 suggestions for Isabelle/HOL and 575 for a select entries from the Archive of Formal Proofs, that could potentially benefit maintainability and readability.

6.1. Future Work

Although the linter is usable in its current state, the goal was to "get the ball rolling" and provide a proof of concept on how a linter for Isabelle might be developed. This seems a successful attempt that provides a solid ground for improvements and further development, such as:

- Applying the suggestions generated by the linter
- Developing more lints
- Exploring more lint types, such as the ones for the inner syntax
- Exploring CI/CD integration
- Expanding the support to Isabelle/VSCode by integrating with the Isabelle/PIDE language server protocol

It might also be interesting to consider further expanding the Isabelle ecosystem, by:

- Developing and using a component that parses the whole Isabelle syntax to an *AST*

- Exploring a formatter or lints that are specialized in how theories should be formatted

Lastly, it is absolutely crucial to interact with the Isabelle community to further evaluate the usability of the linter, get feedback on the lints implemented, and explore what extra features are requested.

A. Lints and Bundles

Currently, the linter has 20 lints implemented, mainly from *Gerwin's Style Guide for Isabelle/HOL* [7, 8]. These are concise descriptions of each of the checks implemented:

Apply-Isar Switch Severity: Medium
Switching form an apply script to a structured Isar proof results in an overall proof that is hard to read without relying on Isabelle. The Isar proof is also sensitive to the output of the apply script, and might therefore break easily. This lints finds such instances.

Auto Structural Composition Severity: Low
The check suggests to replace `apply (auto; ...)`, which might be hard to reason about: structural composition applies the method on the right-hand side only to the new goals generated by `auto`.

Axiomatization with where Severity: High
The `axiomatization` command can potentially introduce inconsistencies into the logic when coupled with a where clause. It should be avoided, unless when used to formalize a logic. This lint detects this problem.

Bad style command Severity: Medium
Detects `back` and `apply_end` commands.

Complex Isar initial method Severity: Medium
Warns about using complex methods in the `proof` command, like `proof auto`, which can make it difficult to read the proof and understanding its goals.

Complex method Severity: Medium
Finds complex methods. A complex method is defined as a method that:

- has more than one modifier (`?`, `+` or `[]`)
- or, has modifiers that are not in the outmost level (e.g. `auto[3] | blast`)

- or, has three or more combinators (`|`, `;` or `,`)

Counter-example finder Severity: High
Detects commands that find counter-examples like `nitpick`.

Diagnostic command Severity: Low
Finds diagnostic commands like `ML_val`, `welcome` or `find_consts`.

Force failure Severity: Low
Some proof methods, like `simp`, might not solve its goal. However, when this is expected, it might be a good idea to combine it with the `fail` method (e.g. `apply (simp; fail)`) in order to force it to fail if the goal is not solved. This facilitates finding where proofs breaks because of changes in the simp set or in the `simp` tactic itself. The lint allows for a way to find these `simp` invocations.

Global attribute changes Severity: Low
Declaring lemmas as `simp` just to be used locally in some some proofs and then deleting them from the simp set is to be avoided, since it affects the global simp set and might make it hard to merge theories together. Instead, these changes should be localized, for example through `context`. This lint detects such declarations.

Global attribute on unnamed lemma Severity: High
Finds unnamed lemmas with the `simp` attribute. This anti-pattern makes it hard to remove this lemma from the simp set when needed.

Implicit rule Severity: Medium
Finds usage of the `rule` method without explicitly stating which rule to use. These invocations might fail when the rule discovery strategy changes, and figuring out which rule worked initially might difficult, making the proof hard to maintain.

Lemma-transforming attribute Severity: Medium
Warns about using the attributes `rule_tac` and `simplified` on lemmas.

Low-level apply chain Severity: Low
Detects long apply scripts that use low-level proof methods (like `rule` or `clarsimp`) and are longer than 5 commands.

Proof-finder Severity: High
Finds proof-finder commands such as `try` and `sledgehammer`.

Short name Severity: Low
Checks for names used for `fun` or `definition` that are shorter than two characters. *Note:* only detects short names that are declared at the outer syntax level.

Unfinished proof Severity: High
Finds `sorry`, `oops` and `\<proof>` commands.

Unrestricted auto Severity: High
Using `auto` in the middle of a proof on all goals (i.e. unrestricted) might produce an unpredictable proof state. It should rather be used as a *terminal* proof method, or be restricted to a set of goals that it fully solves. This lint finds such unrestricted invocations.

Use by Severity: Low
Finds potential `apply` scripts that can be replaced by the `by` command.

Use Isar Severity: Low
Triggers whenever the `apply` command is used, and suggests to use a structured proof instead.

In addition to these checks, one more debugging lint is implemented:

Print AST Severity: Low
Prints the parsed AST.

The lints are grouped into *bundles* to provide presets that can be used to configure the linter. These are:

All Includes all lints.

Default A default set of lints.
Lints included: *Apply-Isar switch, Bad style command, Complex Isar initial method, Complex method, Counter example finder, Global attribute changes, Global attribute on unnamed lemma, Implicit rule, Lemma transforming attribute, Unrestricted auto, Use by*

Foundational A set of lints that can be used while defining a new logic. Similar to the default bundle, but without the `axiomatization_with_where` lint.

Lints included: *Apply-Isar switch, Bad style command, Complex Isar initial method, Complex method, Global attribute changes, Global attribute on unnamed lemma, Implicit rule, Lemma transforming attribute, Unrestricted auto, Use by*

Afp A set of lints that should be used when developing entries for the *Archive of formal proofs.*

Lints included: *Apply-Isar switch, Bad style command, Complex Isar initial method, compleX method, Counter-example finder, Global attribute changes, Global attribute on unnamed lemma, Implicit rule, Lemma transforming attribute, Unrestricted auto, Use by*

Pedantic A set of lints that are too strict.

Lints included: *Auto structural composition, Force failure, Low level apply chain, Short name, Use Isar*

Non interactive A set of lints to disable interactive commands.

Lints included: *Counter example finder, Diagnostic command, Proof finder, Unfinished proof*

B. XML and JSON reporting example

The following are prettified excerpts of the outputs of `isabelle lint` in XML and JSON modes:

```
$ isabelle lint -r json HOL
{
  "reports":[
    ...
    {
      "theory":"HOL.Inductive",
      "report":{
        "results":[
          {
            "name":"use_by",
            "stopPosition":"205:7",
            "stopOffset":6746,
            "edit":{
              "startOffset":6673,
              "stopOffset":6746,
              "replacement":"by (erule gfp_upperbound [THEN subsetD]) (
              erule imageI)",
              "msg":null
            },
            "startOffset":6673,
            "severity":"Low",
            "startPosition":"203:3",
            "commands":[
              -156471,
              -156473,
              -156475
            ]
          }
        ]
      },
      "timing":129
    },
    ...
  ]
}

$ isabelle lint -r xml HOL
<reports>
  ...
  <report theory="HOL.Inductive" timing="129">
      <lint_result lint_name="use_by"
      lint_message="Use &quot;by&quot; instead of a short apply-script."
      lint_severity="Low"
      lint_commands="-111928,-111930,-111932">At <lint_location
           offset="6673"
           end_offset="6746">203:3</lint_location>:
 Use &quot;by&quot; instead of a short apply-script.
    Consider:
          <lint_edit
           offset="6673"
           end_offset="6746"
           content="by (erule gfp_upperbound [THEN subsetD]) (erule imageI)">
            by (erule gfp_upperbound [THEN subsetD]) (erule imageI)
            </lint_edit>
    Name: use_by
    Severity: Low
      </lint_result>
  </report>
  ...
</reports>
```

List of Figures

3.1. Example Syntax 6
3.2. Isabelle/jEdit in action 7

4.1. Architecture Overview 10
4.2. Grammar parsed by the linter 11
4.3. Isabelle/jEdit with linter integration 17
4.4. Output of `isabelle lint` text mode 20

5.1. Linting time (All theories) 24
5.2. Linting time (Theories smaller than 3000 lines) 25

List of Tables

5.1. Lint summary of Isabelle/HOL 23
5.2. Lint summary of the AFP selection 24

Bibliography

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual : Version 6.1*. Research Report RT-0203. Projet COQ. INRIA, May 1997, p. 214.

[2] F. van Doorn, G. Ebner, and R. Y. Lewis. "Maintaining a Library of Formal Mathematics." In: *Lecture Notes in Computer Science*. Springer International Publishing, 2020, pp. 251–267. DOI: `10.1007/978-3-030-53518-6_16`.

[3] *ESLint*. `https://eslint.org/`.

[4] Y. Herklotz, J. Pollard, N. Ramanathan, and J. Wickerson. *Formal Verification of High-Level Synthesis*. 2020.

[5] G. Hutton and E. Meijer. *Monadic parser combinators*. 1996.

[6] S. C. Johnson. *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977.

[7] G. Klein. *Gerwin's Style Guide for Isabelle/HOL. Part 1: Good Proofs.* May 2015.

[8] G. Klein. *Gerwin's Style Guide for Isabelle/HOL. Part 2: Good Style.* May 2015.

[9] N. Mitchell. *HLint*. `https://github.com/ndmitchell/hlint`.

[10] A. Moors, F. Piessens, and M. Odersky. "Parser combinators in Scala." In: *CW Reports* 54 (2008).

[11] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. "The Lean Theorem Prover (System Description)." In: *Automated Deduction - CADE-25*. Springer International Publishing, 2015, pp. 378–388. DOI: `10.1007/978-3-319-21401-6_26`.

[12] U. Norell, N. A. Danielsson, A. Abel, and J. Cockx. *Agda*. `https://github.com/agda/agda`.

[13] *Pylint - code analysis for Python*. `https://www.pylint.org/`.

[14] The mathlib Community. "The lean mathematical library." In: *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. ACM, Jan. 2020. DOI: `10.1145/3372885.3373824`.

[15] Trustworthy Systems Team. *Sel4 Proofs For Api 1.03, Release 2014-08-10*. 2014. DOI: 10.5281/ZENODO.591732.

[16] P. Wadler. "How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages." In: *Conference on Functional Programming Languages and Computer Architecture*. Springer. 1985, pp. 113–128.

[17] M. Wenzel. "Asynchronous Proof Processing with Isabelle/Scala and Isabelle/-jEdit." In: *Electronic Notes in Theoretical Computer Science* 285 (Sept. 2012), pp. 101–114. DOI: 10.1016/j.entcs.2012.06.009.

[18] M. Wenzel. "Interaction with formal mathematical documents in Isabelle/PIDE." In: *International Conference on Intelligent Computer Mathematics*. Springer. 2019, pp. 1–15.

[19] M. Wenzel. *Isabelle/jEdit*. https://isabelle.in.tum.de/doc/system.pdf. Feb. 2021.

[20] M. Wenzel. "Isabelle/PIDE after 10 years of development." In: *UITP workshop: User Interfaces for Theorem Provers. https://sketis. net/wp-content/uploads/2018/08/isabellepide-uitp2018. pdf*. 2018.

[21] M. Wenzel. *The Isabelle system manual*. https://isabelle.in.tum.de/doc/system.pdf. 2021.

[22] M. Wenzel et al. *The isabelle/isar reference manual*. https://isabelle.in.tum.de/doc/isar-ref.pdf. Feb. 2021.

[23] M. Wenzel, S. Berghofer, F. Haftmann, and L. Paulson. *The Isabelle/Isar Implementation*. https://isabelle.in.tum.de/doc/implementation.pdf. Feb. 2021.