Chapter 1

Background

1.1 Linting and Static Analysis

Linting is the process of analysing source code to identify and report issues related to coding style and potential logical errors. The term originates from the `lint` program [**johnson_lint_1978**], which examined C source code for bugs, as well as wasteful code patterns that may be legal but error-prone. The tool was also utilised to enforce portability restrictions which aided users in writing portable code that could be compiled on multiple platforms. Since the release of `lint`, many linting tools, known as **linters**, have been developed for a wide range of programming languages.

Nowadays, many linters can be integrated into IDEs, where code analysis performed by the linter is run incrementally in the background. Any violations found by the linter are displayed directly in the editor as warnings or errors at the relevant locations in the source code. This brings early, real-time feedback to the programmer, allowing them to address issues as they write code, with minimal interruption to their development workflow. Linters can also be integrated as part of the code review process, or into continuous integration (CI) pipelines to ensure that code adheres to a set of standards before being merged into the main codebase.

Although the traditional definition for linting is concerned only with *detecting* issues in code, modern linters have broadened their scope significantly. In addition to detecting issues, many linters provide *auto-fix* capabilities to correct issues by automatically rewriting the offending code snippets. This feature is often integrated into IDEs as well: the popular Language Server Protocol for defining IDE features enables these auto-fix features via *code actions* [**gunasinghe_lsp_2022**]. When a section of code is highlighted by a linter warning, a user can apply a code action to automatically fix the issue with a single click.

Linters and related static analysis tools are increasingly becoming more important in modern software development, as modern code continues to become more complex and difficult to reason about. Industry leaders, such as Google [**sadowski_analysis-google_2018**] and Meta/Facebook [**calcagno_moving-facebook_2015**], have embraced these tools as integral components of their software development workflows. The use of automated tools to detect potential issues is not only faster but in some cases more effective than human review, saving developer time and reducing error rates in the development process.

1.1.1 Types of Issues Detected by Linters

Many linters are configurable with a set of rules, which specify the categories of issues that the linter should detect. These rules can be enabled or disabled by users, allowing them to customise the linter to their needs. Rules are usually grouped by purpose: some rules are concerned with simply improving code style, while others are concerned with detecting suspicious code patterns indicative of potential bugs.

Style checks and code quality

Linters can suggest opportunities to improve code by utilising language features in a more idiomatic manner. Snippets of code that violate these stylistic rules are not necessarily incorrect, but should be fixed as they may be harder to read or maintain in the long term. Furthermore, many idiomatic practices exist to avoid common pitfalls that could lead to unintended behaviour. By highlighting good practices, linters can help users avoid these common mistakes that may cause bugs. For example, *ESLint*[1], one of the most popular JavaScript linters, warns against common JavaScript pitfalls such as using the regular equality operator == instead of its type-safe alternative ===.

A well-designed linter can help programmers learn about useful language constructs by suggesting them in the context of their code, aiding them in adhering to best practices and common style conventions. This category of rules is therefore especially helpful as an educational tool for new users of a language, who may be unaware of these idioms. For example, the *Clippy*[2] linter for Rust [**li_clippy_2023**] categorises a collection of rules as `clippy::complexity` rules to detect code that does something simple in a complex way and suggests a simpler alternative. **??** provides an example of a similar rule in Haskell, from the *HLint*[3] linter. The rule suggests to rewrite the function into an equivalent but more concise form via η-reduction, presented to the user as a code action that can be applied automatically.

```
foo xs = map (+1) xs
```

(a) A Haskell function `foo`, which can be made more concise using η-reduction.

```
Eta reduce
Found:
  foo xs = map (+ 1) xs
Why not:
  foo = map (+ 1)
hlint(refact:Eta reduce)
```

(b) The linter warning shown for `foo`.

Fig. 1.1: An example of a warning from the Haskell linter `hlint`, suggesting a fix that a user can choose to automatically apply.

Domain-specific idioms A library or especially an embedded DSL may require a particular style of usage that is different from the host language [**hora_domain_2012**]. The majority of linters are designed for general-purpose application domains, so they are unlikely to detect issues specific to a more specialised domain. Therefore, linters may be developed for specific libraries or DSLs, with their own set of domain-specific rules. In this case, the accompanying linter can benefit users and improve developer productivity in a similar manner to general-purpose linters: common misuses can be detected and sometimes automatically fixed, and users can be directed to relevant documentation to learn more about correct usage. For instance, the *xUnit.net* testing framework for C# is accompanied by the `xunit.analyzers`[4] package which provides linting rules to enforce best practices specific to *xUnit*.

[1] https://eslint.org/docs/latest/rules/
[2] https://doc.rust-lang.org/clippy/
[3] https://hackage.haskell.org/package/hlint
[4] https://github.com/xunit/xunit.analyzers