Programming Language Tooling for Hylo

Incremental Compilation and Analysis

Dobrin Bashev[1]
Supervisor(s): Andreea Costea[1]**, Jaro Reinders**[1]
[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 22, 2025

Name of the student: Dobrin Bashev
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Jaro Reinders, Harm Griffioen

Abstract

This paper presents a conceptual model for enabling incremental compilation in Hylo, a modern memory-safe systems programming language currently under development. Drawing from a comparative study of existing incremental compilation techniques, an in-depth analysis of Hylo's front-end architecture, and exploratory experiments, the model categorizes common types of program changes, introduces a scope-aware AST diffing algorithm, and proposes a lightweight dependency tracking mechanism tailored to Hylo's needs.

1 Introduction

Efficient compilation is a fundamental requirement for modern programming languages, particularly as software systems scale in both size and complexity. In today's development environments, rapid iteration and immediate feedback are vital to maintaining developer productivity and sustaining momentum in software projects. Long compilation times introduce friction into the development process, interrupting workflows and potentially discouraging experimentation or refactoring. Incremental compilation - where only the changed portions of a program are recompiled - has emerged as a key strategy to mitigate this issue [1]. This approach reduces unnecessary recomputation, allowing developers to benefit from faster build times even when they are working on large and interdependent codebases.

Beyond the build process itself, the need for responsiveness also influences the design of modern development tools such as language servers. Highly responsive language servers must be capable of analyzing source code in real time, offering features like code completion, diagnostics, and refactoring tools with minimal latency. Achieving this level of interactivity requires sophisticated internal designs that can mirror some of the principles of incremental compilation - for example, reusing previous analysis results, managing fine-grained dependencies, and reacting efficiently to partial changes in code. As a result, both compilation strategies and development tooling have evolved to meet the expectations of modern software engineering workflows.

Hylo is a relatively new systems programming language that prioritizes performance and safety. The focus on safety in particular is achieved through adopting mutable value semantics [2] and incorporating expressive generic systems [3], both of which aim to prevent many subtle programming mistakes. However, Hylo's compiler currently lacks extensive support for incremental compilation which could limit its future adoption in real-world development environments. While the compilers of languages with a similar profile like Swift and Rust have been extensively optimized for incremental behavior, their techniques are not directly applicable to Hylo due to its unique language semantics and compiler infrastructure. This leaves a gap in both practical tooling and academic understanding of how incremental compilation can be adapted to newer, safety-focused languages. Advancing research and tooling in this area - through comparisons to other languages and improvements in the compiler design - could close this gap and bring Hylo's development experience closer to that of more mature systems languages.

The core research question of this project is to investigate how existing incremental compilation techniques can be effectively adapted and optimized for the Hylo programming language. This work answers that question by splitting it into three subquestions:

- RQ1: What are the current incremental compilation techniques, and how have they been applied in modern programming languages?

- RQ2: How are the semantics of the Hylo language defined, and what are the key architectural features of the Hylo compiler front-end?
- RQ3: Which techniques are most suitable for integration into the Hylo compiler, and how could such techniques be prototyped and evaluated?

By executing a comparative analysis of existing tooling approaches, designing a prototype model tailored to the specific needs of Hylo, and assessing their applicability and the key challenges associated with their implementation, this paper aims to lay the foundations for more performant and adaptable language tooling in modern programming ecosystems.

2 Background

This section provides essential context for understanding incremental compilation in Hylo. It begins by outlining the Hylo compilation model, which establishes the foundation of how Hylo code is processed. This is followed by an overview of query-based compilation, a technique that enables efficient recomputation by modeling compilation as a set of interdependent queries. Lastly, the section discusses the granularity of compilation units, highlighting how the size and scope of compilation units impact the effectiveness of incremental updates.

2.1 Hylo compilation model

In the Hylo compilation model [4], the front-end begins by parsing each source file separately, in a sequential manner. During parsing, the compiler breaks down the raw code into a detailed abstract syntax tree (AST) for the whole module. This process separates the program's syntactic structures into distinct categories - statements, declarations, patterns, and expressions - capturing all necessary information for each. This rich representation allows later stages to precisely understand and manipulate the program's components.

Following parsing, the compiler constructs a comprehensive scope hierarchy spanning the entire module. This step is performed partially in parallel for all files and focuses on organizing declarations into nested scopes that define where identifiers can be declared and accessed. Rather than resolving names at this point, the scoping phase sets up the lexical framework of visibility rulesv.

Full name resolution happens on demand during the typing phase, where the compiler treats the whole module as one unit and queries the name resolution system as needed. Typing - covering both type-checking and type-inference - depends heavily on this on-the-fly resolution to ensure all names are correctly bound and all types are consistent. The expressiveness of Hylo's generic system necessitates the use of constraint solving algorithm to correctly identify the types referred in each context. The model of the front-end is illustrated in **Figure 1**.

After successful typing, the back-end takes the fully annotated AST and emits the Hylo immediate representation (IR). This stage effectively links the modules together by integrating them one by one into the generated code for the whole program. Finally, the Hylo IR is transpiled to the LLVM IR and LLVM is invoked for the last step of the process - the generation of the machine code. The model of the back-end is illustrated in **Figure 2**.

It is important to note that Hylo's compiler has limited support for incremental compilation in the form of module-level caching of the results produced by the front-end. In other words, once parsed and typed the AST for a given module can be reused in the subsequent

stages of the compilation of various programs which use it. A prominent example of that is Hylo's standard library.

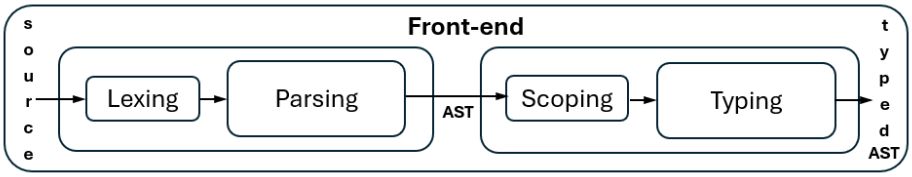

Figure 1: Scheme of Hylo's front-end model

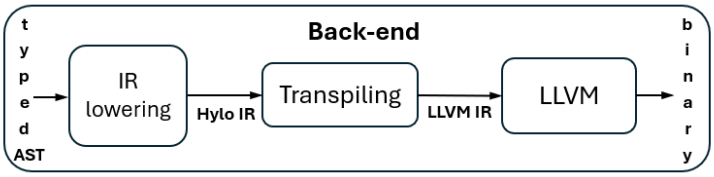

Figure 2: Scheme of Hylo's back-end model

2.2 Query-based compilation

The query-based model of compilation represents a sophisticated approach to incremental compilation, designed to optimize build times by intelligently reusing past computation [5]. This model fundamentally re-architects the compiler's internal workings around a system of "queries", which are essentially pure functions meaning that given the same inputs, will always produce the same output. This purity is crucial because it allows the compiler to reliably cache the results of these queries and efficiently invalidate them when a relevant change is introduced.

The compilation process is broken down into numerous, often fine-grained, tasks, each represented by a query. For instance, a query might compute the AST for a specific file, perform type checking for a function, or resolve a declaration's value. The results of these individual queries are then cached. When a subsequent compilation run occurs, the system first checks if the inputs to a query have changed. If they haven't, the cached result can be reused, avoiding redundant computation. This caching mechanism is a primary driver of efficiency in incremental builds, significantly reducing the "from-scratch" compilation time. Modern IDEs, for example, often hold intermediate products in memory to avoid costly re-parsing and loading, enabling faster feedback [6].

Some queries may need access to information from other, potentially distant, parts of the program's analysis. Therefore, a critical aspect of the query-based model is the explicit and automated tracking of dependencies between these queries [7]. As one query invokes or accesses the result of another query, this interaction is recorded, building a dependency graph. This graph is inherently acyclic (a directed acyclic graph or DAG), ensuring a resolvable evaluation order. For example, in Rustc's incremental compilation system, dependencies are recorded between internal "queries", making these relationships explicit [8]. This explicit

dependency tracking allows the compiler to understand precisely what data contributed to a query's result by following the edges of the graph.

When a change occurs in the source code, the system needs to determine which cached query results have been invalidated and thus require re-evaluation. This is typically achieved through a combination of change detection and a propagation algorithm. Once a change is detected, a propagation algorithm, such as the "red-green algorithm", is employed [9]. Nodes (query results) in the dependency graph are initially marked "red" if they are potentially affected by a change. The system then attempts to re-evaluate these "red" queries. If a re-evaluated query produces the same result as its cached version (even if its inputs changed), it can be marked "green", and its dependents do not need to be re-evaluated. If the result has changed, it remains "red", and its dependents are then considered for re-evaluation. This interleaving of change detection and re-evaluation helps mitigate "false positives", where a query might be re-evaluated only to find its output hasn't actually changed, avoiding unnecessary downstream recompilations.

2.3 Granularity of compilation units

A significant design consideration in the query-based model is the granularity of the compilation units (the "queries"). Smits et al. discuss this fundamental trade-off in [1]:

Smaller Units (Fine-grained): Breaking the compilation into very small, fine-grained queries (e.g., function-level, statement-level, or even specific interface elements) allows for highly precise dependency tracking. When a minor change occurs, only a minimal set of these small units and their direct dependents need to be re-evaluated. This leads to faster incremental builds because less unnecessary work is performed. However, managing a vast number of small units introduces more dependency relations, increasing the overhead of graph construction, storage, and traversal. The cost of computing fingerprints for many small units can also be substantial.

Larger Units (Coarse-grained): Conversely, using larger, coarse-grained units (e.g., file-level or module-level compilation) simplifies the dependency graph, as there are fewer nodes and edges to manage. This can reduce the overhead associated with dependency tracking itself. However, a change within a large unit might necessitate re-evaluating the entire unit, even if only a small part was affected. This can lead to over-approximation of dependencies and more time spent re-evaluating the larger unit than strictly necessary, potentially negating the benefits of incrementality for small changes.

The optimal balance lies in choosing a granularity that minimizes the total cost of compilation, considering both the overhead of dependency management and the time spent on re-evaluation. Modern systems often strive for fine-grained dependencies to maximize efficiency, especially in interactive development environments [6].

3 Methodology

This research followed an exploratory approach to assess the feasibility of integrating incremental compilation into the Hylo programming language. A combination of methods - including literature review, source code analysis, expert consultation, and compiler experiments - was used to investigate the language's architecture and existing compilation process. These are described in subsection 3.1.

Early in the project, a conceptual model for incremental compilation in Hylo was developed. This model served as a basis for the main part of the research process and provided

a framework for evaluating technical limitations and identifying areas for improvement. It is explained in subsection 3.2.

3.1 Research methods

To investigate the feasibility of integrating incremental compilation into the Hylo programming language, a mixed-method approach was adopted. This involved a combination of theoretical research, direct interaction with the compiler's codebase and developers, and exploratory experiments aimed at identifying architectural constraints and informing the design of a conceptual solution.

- Literature review and comparative analysis: Conducted an in-depth survey of academic literature, technical documentation, and existing compiler architectures to identify and analyze established techniques for incremental compilation. This provided a conceptual foundation and a set of comparative results to guide the adaptation of such techniques to Hylo's unique architecture.
- Code review and expert consultation: Closely examined the front-end source code of the Hylo compiler to gain a deep understanding of its parsing, desugaring, and semantic analysis phases. Engaged in technical discussions with Hylo developers Dimitri Racordon and Ambrus Toth to clarify the rationale behind key design decisions, uncover undocumented behaviors, and map out the internal compilation workflow.
- Experiments and conceptual design: Conducted exploratory experiments with the Hylo compiler to better understand its behavior, internal architecture, and limitations in the context of incremental compilation. These experiments involved modifying and observing the compiler in controlled scenarios to identify potential shortcomings and areas for improvement. Based on these findings, developed a high-level conceptual model outlining how incremental compilation could be integrated into Hylo. No implementation of incremental techniques was performed; the focus remained on feasibility analysis and architectural planning.

3.2 Conceptual model

The conceptual model for incremental compilation in Hylo was developed in the early stage of the project as a theoretical framework to guide subsequent investigation. It emerged from initial experiments with the compiler and technical consultations with core Hylo developers. These early findings highlighted the front-end of the compiler as the most promising candidate for incrementalization.

In the experiments, a synthetic Hylo program, which contains relatively simple language features, was generated. The program consisted of 100 structures, each of which conformed to one of three traits requiring implementations for up to 3 methods (see **Listing 1** in Appendix for details). As a result, the parsing and scoping stages were found to be sufficiently fast to justify rerunning them in full on each compilation. The typing phase was found to be more than 20 times slower than parsing and more than 30 times slower than scoping even in such non-complicated situations. Consequently, it was established that the model should focus on incrementalizing the typing phase.

The proposed model focuses on tracking and responding to changes at the level of the abstract syntax tree (AST), enabling selective re-analysis of modified components in the compilation pipeline. It consists of two main components: a mechanism for computing

differences between successive ASTs, and an incremental approach to type checking that limits recomputation to affected subtrees. An illustration of it can be found in **Figure 3**.

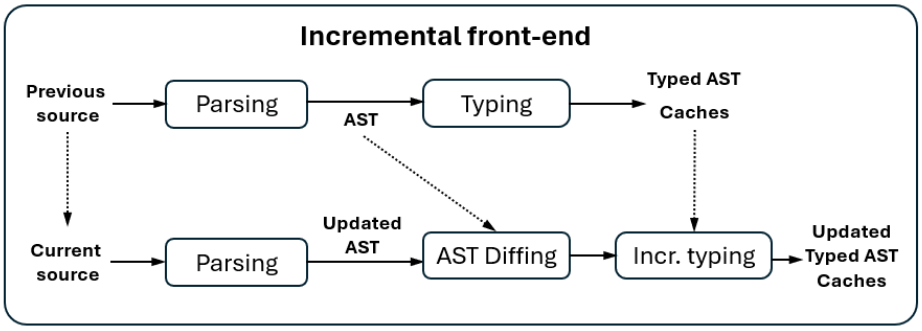

Figure 3: Scheme of conceptual model for incremental front-end

The two main ideas of the model are outlined in the following subsections.

3.2.1 AST diffing

Abstract Syntax Tree (AST) diffing [10] enables a structured comparison between two versions of a program by identifying the precise syntactic and semantic changes between their tree representations. Rather than relying on raw textual differences, AST diffing captures meaningful transformations such as changes in function signatures, addition or removal of definitions, or the reordering of declarations - by aligning nodes and labeling edits (insert, delete, update). In the context of Hylo, AST diffing can be used to systematically detect and classify edits made between program revisions. This structured change model provides a foundation for tools to understand not only what changed, but how those changes impact the rest of the program's semantics.

3.2.2 Incremental approach to type checking

By leveraging the results of AST diffing, Hylo's type checker could be incrementalized, avoiding full re-typechecking of the entire program on every change. When an edit is made, the AST diff indicates the minimal scope of change, allowing the type system to invalidate only those parts of the type environment and dependency graph that are directly affected. For example, if a function's return type is altered, only its call sites and dependents need rechecking. Combined with caching and dependency tracking, this localized re-typechecking process has the potential to dramatically reduce latency during development, improving responsiveness in IDEs or build systems. Thus, AST diffing serves as the backbone of a precise invalidation strategy, which would enable Hylo to deliver both strong static guarantees and interactive performance at scale.

4 Incremental Compilation in Hylo

This section presents the core results of the research, outlining a high-level model for integrating incremental compilation into the Hylo compiler. It begins by classifying common types of program changes and assessing their impact on the compilation process. Building

on this, it introduces an AST diffing strategy tailored to Hylo's syntax and semantics, addressing the limitations of naive index-based comparisons. Finally, it proposes a lightweight dependency tracking mechanism designed to support precise and efficient invalidation during recompilation.

4.1 Classification of program changes

To design an efficient incremental compilation system, it is crucial to classify the types of program changes according to their semantic impact and the extent to which they propagate through the compilation pipeline. In Hylo, changes can be broadly categorized based on their visibility, disruptiveness, and detectability.

4.1.1 Implementation changes to functions

The most frequent type of change encountered in day-to-day programming is a modification of a function's implementation. In Hylo, methods are treated as syntactic sugar over functions with an implicit self parameter, so both fall under the same category semantically. These changes typically involve editing the function body without altering its signature (i.e., parameter types, return type, or labels). As such, they are entirely local to the function itself and invisible to the rest of the program from the type-checking perspective.

Good coding practices in Hylo (and in general) encourage short and well-encapsulated function bodies, which further limits the cost of reanalyzing such changes. Given that the function's interface remains unchanged, re-type-checking its body in isolation is both feasible and sufficient, making this class of changes the most straightforward to support incrementally.

4.1.2 Declaration reordering

Another class of benign changes includes reordering declarations within a single scope where order is semantically irrelevant. In Hylo, this applies to the top-level declaration as well as those introduced in the bodies of trait, extension, and struct declarations, where the relative position of members does not affect their meaning. When such reorderings occur, the abstract syntax tree (AST) remains structurally identical in terms of typing dependencies.

Since no identifiers are added, removed, or semantically altered, and lookup mechanisms do not depend on declaration order in these contexts, these changes can be entirely ignored by the type-checker. As a result, declaration reordering within reorderable scopes can be treated as a no-op for incremental compilation.

4.1.3 Identifier renaming

Renaming an identifier may seem like a purely cosmetic transformation, but it introduces subtle semantic and structural challenges. In Hylo, names of function parameters and record members are significant - for instance, changing `fun f(param1: inout A)` to `fun f(param2: inout A)` defines a distinct function [11]. Similarly, {a: {}} and `{b: {}}`, despite structural similarity, are not interchangeable. Therefore, renaming in these contexts must be treated as replacing one declaration with a new, unrelated one.

The only safe and incremental-friendly form of renaming occurs when all references to an identifier are simultaneously updated. This can typically be guaranteed only through automated refactoring tools like a language server would normally support. Until such

tooling is developed for Hylo, renames must conservatively be treated as a deletion followed by an introduction, potentially triggering full reanalysis of affected scopes.

4.1.4 Public API changes

Changes to the public interface of declarations are the most disruptive to incremental compilation. They require careful dependency tracking due to their potentially wide-reaching impact. Two strategies are possible:

- Conservative invalidation: This approach invalidates all caches for scopes that are transitively visible from the changed declaration. Although simple to implement, it is coarse-grained and often inefficient. Even the addition of an unused top-level function would cause unnecessary recompilation across unrelated modules.
- Dependency graph-based invalidation: A more precise strategy involves tracking references to each declaration and using them to construct a dependency graph. When a declaration's public API changes, only the nodes (i.e., dependent scopes) reachable via this graph are invalidated. This requires maintaining up-to-date and fine-grained reference information but enables more targeted and efficient incremental recompilation.

4.2 AST diffing approach

To enable effective incremental compilation in Hylo, the compiler must be able to detect which parts of a program have changed following a code edit. This is accomplished through AST diffing - a process of comparing the abstract syntax tree generated before the change with the one generated after parsing the updated program. The goal of this process is to identify and isolate the minimal set of modified constructs so that only the affected parts of the program are re-analyzed.

4.2.1 Limitations of index-based comparisons

The straightforward comparison of ASTs is complicated by the fact that the AST nodes are not stable across parses. Internally, each AST node has an identity which is a convenient umbrella representation for the index of the module, the index of the file in the modules and the index of the node in the corresponding file. Since these indices are given in the order they are parsed, even a small edit - such as inserting a new declaration or modifying a function body - can completely change the identities of the subsequent nodes. Consequently, relying on these identities for detecting changes is infeasible and misleading, as even unchanged parts of the program may appear different merely due to index shifts.

Furthermore, syntactic changes like reordering declarations, inserting comments, or formatting adjustments may not affect semantics at all, yet they can obscure the correspondence between the old and new ASTs if structural information is not taken into account.

4.2.2 Hierarchical scope-based diffing algorithm

To address this, we can use a hierarchical and identifier-aware diffing approach, designed to find matches even in case of identity mismatch. The steps are the following:

1. Matching declarations by identifiers

   The diffing process starts at the outermost level of the program structure - the top-level of the files in the module. We attempt to match declarations (such as structs, traits, extensions and givens) in the old and new ASTs on the basis of the signature which includes the identifier and the other relevant attributes (such as type annotations, access modifiers, conformances in case of structs or parameters in case of functions). For each declaration encountered, we determine one of the following cases:

   - The declaration is newly introduced (present in the new AST but not in the old one).
   - The declaration no longer exists (present in the old AST but missing from the new one).
   - The declaration continues to exist (present in both ASTs, with matching identifiers).

   In the first two cases, we keep track of the changes associated to the corresponding scope.

   Matching declarations by name provides a stable anchor point even in the presence of unrelated edits elsewhere in the file or module.

2. Comparing declarations within matching scopes

   For declarations that persist between the two versions, we then recursively compare the sets of declarations contained in their inner scopes. The exception is made for function declarations which are the base case for the recursion. If function declaration is newly introduced or its body has changed, then we will analyze it again because of the reasons explained in **section 4.1.1**. Otherwise, we will try to use as much of the results of the previous analysis as possible.

   To decide efficiently whether the body of a function has changed, we use hashes which are computed during the parsing. The size of the hashes should be at least 128 bits to make the risk of collision negligible. [9]

3. Pruning unchanged branches of the AST

   To further optimize the traversal of the AST, we can again apply hashing to identify whether we are entering a branch in the AST which has not changed at all. However, as we have already discussed, the order of the declarations matters only in function scopes, which means that we cannot directly apply hashing without allowing unnecessary work in case of benign re-orderings. There are two ways to solve this problem:

   - Sorting declarations inside each scope with stable ordering before hashing: By sorting the declarations in scopes where order does not matter (such as module-level or struct-level scopes), we can ensure that semantically equivalent but syntactically reordered code produces the same hash. A stable sort guarantees that declarations that are already in order are not arbitrarily moved, preserving relative order among equal elements where needed. This approach allows the hashing function to treat benign reordering as unchanged code, thus avoiding redundant traversal and reanalysis.

- Using special hash functions that are position-independent: Alternatively, we can design or use hash functions that are inherently insensitive to the order of items within a collection. For instance, instead of hashing a list of declarations as a sequence, we can hash the multiset of declarations, which ignores their order. These kinds of hash functions can treat two declaration sets with the same elements but in different orders as equivalent, provided the semantic meaning is unaffected by the order (which is the case outside function bodies).

By applying these strategies, we can minimize unnecessary work and focus reanalysis efforts only on genuinely modified parts of the AST, significantly improving the performance of incremental compilation or transformation tools.

For pseudo code of this algorithm see **Listing 2** in the Appendix.

4.3 Dependency tracking

During the typing phase of the Hylo compiler, dependency tracking can be implemented by associating each declaration with a set of functions that depend on it. This is achieved by recording, during the typing of a function body, the declarations resolved through name lookup - these can be types used in annotations or functions invoked within the body. For every such resolved declaration, the compiler updates its internal metadata to include a reference to the current method as one of its dependents.

This reverse mapping - from declaration to dependent methods - enables efficient invalidation when changes occur elsewhere in the codebase. When the AST diff signals that a declaration has been added or removed, the compiler performs a name lookup from the scope of the change. This lookup determines whether any existing declarations have been shadowed or become visible as a result of the change. Both situations imply that a previously resolved name would now resolve to a different declaration, altering the meaning of code that depended on the original.

For each declaration identified as being shadowed in this way, the compiler consults its dependency map to retrieve all methods that previously referred to it. These methods are then marked for reanalysis, as their typing results may no longer be valid under the new resolution. This strategy allows the compiler to selectively re-type only the affected methods, ensuring correctness while maximizing reuse of cached typing results for unaffected code.

5 Discussion

To contextualize the proposed incremental compilation model for Hylo, this section first examines related work by outlining the approaches taken by established compilers such as Swift and Rust. Following that, it discusses the limitations of the current proposal, particularly in light of its reliance on Hylo's evolving compiler infrastructure and the potential challenges involved in translating the conceptual model into a practical implementation.

5.1 Related work

This section presents details about the incremental features incorporated in the compilers of Swift and Rust. Since these languages have similar profile to Hylo, their compiler's inner workings were a significant focus in the early stages of the research process.

5.1.1 Rust's compiler

Rust's compiler (rustc) supports incremental compilation primarily through fine-grained dependency tracking at the module and item level [9]. This is achieved using the incremental compilation engine introduced in the MIR (Mid-level Intermediate Representation) phase.

- Parsing: Rust does not incrementally parse source files. Each modified source file is re-parsed in full. However, unmodified files are skipped entirely.
- Semantic Analysis: The compiler uses metadata and dependency graphs to track definitions and references across crates. When a function or type changes, only the dependent items are re-analyzed. This is facilitated by the Query System, which memoizes the results of compiler queries (e.g., type checking a function) and invalidates them only when their inputs change.
- Optimization: Rust performs MIR optimizations incrementally. The MIR is stored in a cache and reused when possible. Only the affected MIR units are re-optimized upon changes.
- Code Generation: Code generation (LLVM backend) is incremental at the codegen unit level. Rust splits crates into multiple codegen units (CGUs), and only the modified CGUs are recompiled and re-linked. This significantly reduces recompilation time, especially for large projects.

The entire incremental process relies on fingerprinting and hashing of inputs and outputs, stored in an on-disk cache under the target/incremental directory.

5.1.2 Swift's compiler

Swift's compiler also supports incremental compilation, though its model is centered around source file boundaries and leverages driver-level orchestration rather than fine-grained item-level tracking [12].

- Parsing: Swift recompiles and re-parses only the source files that have been modified. This is made possible by file-level dependency tracking, which determines which files depend on which others .
- Semantic Analysis: Like Rust, Swift employs a dependency graph to track inter-file dependencies. However, it performs semantic analysis (e.g., name resolution, type checking) only for changed files and their dependents. The granularity is coarser compared to Rust's item-level tracking.
- Optimization: Swift uses the LLVM optimizer, and optimizations are applied at the module or whole-program level depending on the build configuration. For debug builds, the optimizer is minimal, so incremental recompilation is faster. For release builds, the entire module may be re-optimized unless whole-module optimization (WMO) is disabled.
- Code Generation: Swift emits object files for individual source files. Only the changed files are recompiled into object files, and the linker stitches them together. This object-file-based strategy is analogous to traditional C/C++ build systems.

Swift's incremental builds are coordinated by the Swift driver, which determines which files need to be rebuilt based on timestamps, module interfaces, and a persistent dependency file (.swiftdeps).

In summary, Rust offers more fine-grained incrementality, especially in semantic analysis and MIR-based optimization, using a sophisticated query-based model. In contrast, Swift provides file-level incrementality, offering faster feedback loops with a simpler but coarser approach. The trade-off is between the granularity of change detection and the complexity of the compiler infrastructure required to support it.

With this in mind, the incremental compilation model proposed for Hylo combines characteristics of both Rust and Swift. Like Rust, it avoids incrementally parsing source files, as the potential performance gains do not justify the added implementation complexity. For semantic analysis, Hylo adopts an approach more similar to Swift's, relying on explicit dependency tracking rather than a query-based system. This avoids the additional indirection introduced by Rust's query model while still enabling effective incremental analysis.

5.2 Limitations

While the proposed incremental compilation model for Hylo presents a conceptually solid and pragmatic approach, several limitations must be acknowledged:

1. Implementation-dependent adjustments: Although the model is carefully designed with practical trade-offs in mind, its theoretical soundness does not guarantee seamless integration into the existing compiler. During implementation, unforeseen complexities may arise, necessitating deviations or refinements to the model. Performance bottlenecks, interface constraints, or interactions with other compiler subsystems may lead to compromises in the design.

2. Dependence on the current compiler architecture: The proposed model is closely aligned with the present structure and behavior of the Hylo front-end. However, compiler development is an evolving process, and significant changes to the front-end - such as modifications to the AST, type system, or module resolution mechanisms - could render parts of the model obsolete or incompatible. As such, future adaptations may be required to maintain coherence with ongoing infrastructure changes.

These limitations highlight the importance of treating the current proposal as a flexible framework rather than a fixed implementation plan. Continued evaluation during the development process will be necessary to ensure that the model remains robust and relevant as the compiler matures.

6 Responsible Research

This research has been conducted with a focus on methodological transparency and reproducibility. All methods employed are grounded in established academic practices. The literature review and comparative analysis relied on publicly available academic papers, compiler documentation, and open-source codebases, ensuring that the sources used are accessible for verification and further inquiry. In terms of reproducibility, the methodological approach is designed to be repeatable by others. The Hylo compiler is open-source, and the exploratory experiments described - though informal and qualitative in nature - were

performed using publicly available versions of the compiler. Therefore, all observations and conclusions are based on source-level inspection and documented behaviors. The conceptual design is derived from reproducible steps: analysis of the compiler's behavior, consultation of public resources, and synthesis based on observable properties.

In addition to ethical and reproducibility considerations, this work contributes to a broader security-conscious ethos by focusing on memory-safe programming languages like Hylo. According to the February 26, 2024 US government report Back to the Building Blocks: A Path Toward Secure and Measurable Software, adopting memory-safe languages is a strategic measure to prevent entire classes of vulnerabilities at scale - an approach the report directly encourages technical and academic communities to pursue [13]. By advancing research in safe-by-design compiler architectures, we align with public policy goals aimed at reducing cybersecurity risks at their root and promoting measurable software quality.

7 Conclusions and Future Work

This paper set out to explore how incremental compilation could be effectively introduced into the Hylo programming language. The central research question was: *What would an incremental compilation model tailored to Hylo's architecture look like, and what are the key design challenges involved in building it?*

To address this, we conducted a comparative analysis of existing approaches to incremental compilation, examined the internal structure of Hylo's front-end, and carried out exploratory experiments to understand how typical source-level changes affect the compilation process. The resulting conceptual model introduces three main components:

1. a classification of relevant code changes and their expected impact;
2. a hierarchical, scope-aware AST diffing algorithm to detect and isolate these changes efficiently; and
3. a lightweight dependency tracking mechanism to determine the set of elements that must be recompiled.

Together, these contributions lay a foundation for building more performant and adaptable tooling in modern language ecosystems. A key strength of the proposed model is that it is designed to integrate smoothly with Hylo's existing front-end, avoiding unnecessary complexity while enabling meaningful compiler responsiveness improvements.

However, this work remains conceptual. The next steps involve implementing the proposed techniques within the Hylo compiler, validating their performance impact, and refining the model based on practical experience.

In conclusion, this research provides a structured, implementation-aware approach to incremental compilation for Hylo and sets the stage for future advances in compiler performance and tooling responsiveness in memory-safe language ecosystems.

References

[1] Jeff Smits, Gabriel D.P. Konat, and Eelco Visser. Constructing hybrid incremental compilers for cross-module extensibility with an internal build system. *The Art, Science, and Engineering of Programming*, 4(3), February 2020.

[2] Dimitri Racordon, Denys Shabalin, Daniel Zheng, Dave Abrahams, and Brennan Saeta. Implementation strategies for mutable value semantics. *Journal of Object Technology*, 21(2):2:1–11, 2022. ECOOP 2021 Workshops.

[3] Hylo Project Contributors. Generic environments. `https://github.com/hylo-lang/hylo/blob/main/Docs/GenericEnvironments.md`. Originally created in 2023. Last revised in 2024. Accessed: 2025-06-01.

[4] Hylo Project Contributors. Compiler architecture. `https://github.com/hylo-lang/hylo/blob/main/Docs/Compiler/CompilerArchitecture.md`. Originally created in 2023. Last revised in 2024. Accessed: 2025-06-01.

[5] Sébastien Doeraene and Tobias Schlatter. Parallel incremental whole-program optimizations for scala.js. *SIGPLAN Not.*, 51(10):59–73, October 2016.

[6] Edward Z. Yang. Optimizing incremental compilation. `https://blog.ezyang.com/2016/08/optimizing-incremental-compilation/`, August 2016. Accessed: 2025-06-10.

[7] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. Scalable incremental building with dynamic task dependencies. pages 76–86, 2018.

[8] Michael Woerister. Incremental compilation. `https://blog.rust-lang.org/2016/09/08/incremental/`, September 2016. Accessed: 2025-06-01.

[9] The Rust Project Developers. Incremental compilation in detail. `https://rustc-dev-guide.rust-lang.org/queries/incremental-compilation-in-detail.html`. Accessed: 2025-06-01.

[10] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[11] Hylo Project Contributors. Hylo language specification. `https://github.com/hylo-lang/specification/blob/main/spec.md`. Originally created in 2022. Last revised in 2024. Accessed: 2025-06-10.

[12] Swift Project Contributors. The swift driver, compilation model, and command-line experience. `https://github.com/swiftlang/swift/blob/main/docs/Driver.md`. Originally created in 2016. Last revised in 2024. Accessed: 2025-06-01.

[13] Office of the National Cyber Director. Back to the building blocks: A path toward secure and measurable software. Technical Report ONCD-TR-2024-02-26, White House Office of the National Cyber Director, Washington, DC, February 2024.

A Appendix

```cpp
#include <fstream>

std::ofstream test("prog.hylo");

std::string traits[] =
{
    "trait T1 { fun f() }",
    "trait T2<T> { fun g() -> T }",
    "trait T3<U, V> { fun h(x: U) -> V}"
};

std::string conformances[] = {"T1", "T2<Void>", "T3<Void, Void>"};

std::string methods[] =
{
    "public fun f() {}",
    "public fun g() { self.f() }",
    "public fun h(x: Void) { self.g() }"
};

int main()
{
    int n = 100;

    std::string prog = traits[0] + "\n" + traits[1] + "\n" + traits[2]
        + "\n\n";

    for (int i = 0; i < n; ++ i)
    {
        std::string no = "  ";
        no[0] = i / 10 + 48;
        no[1] = i % 10 + 48;

        prog += "struct S" + no + " is " + conformances[0];
        if (i % 3 > 0) prog += " & " + conformances[1];
        if (i % 3 > 1) prog += " & " + conformances[2];
        prog += " {\n";

        prog += "  " + methods[0];
        if (i % 3 > 0) prog += "\n  " + methods[1];
        if (i % 3 > 1) prog += "\n  " + methods[2];
        prog += "\n}\n";
    }

    test << prog;
    return 0;
}
```

Listing 1: Generation of a synthetic Hylo program (gen.cpp)

```
function diff_modules(old_ast, new_ast):
    // The top-level scope of the program
    let old_top_level_scope = old_ast.top_level
    let new_top_level_scope = new_ast.top_level

    // Initialize a list to store the identified changes
    let changes = []

    // Begin the recursive diffing process from the top-level scope
    diff_scopes(old_top_level_scope, new_top_level_scope, changes)

    return changes

function diff_scopes(old_scope, new_scope, changes):
    // Optimization: Prune unchanged branches using hash comparison
    if calculate_scope_hash(old_scope) == calculate_scope_hash(
        new_scope):
        return // Scopes are identical, no further analysis needed

    // 1. Matching declarations by identifiers
    let old_declarations = create_map_of_declarations(old_scope)
    let new_declarations = create_map_of_declarations(new_scope)

    // Identify continuing declarations
    for each identifier in old_declarations.keys():
        if identifier in new_declarations.keys():
            let old_decl = old_declarations[identifier]
            let new_decl = new_declarations[identifier]

            // 2. Comparing declarations within matching scopes
            if is_function(old_decl):
                // Base case: Handle function declarations
                if has_function_body_changed(old_decl, new_decl):
                    add_change(changes, "MODIFIED", new_decl)
                // else: Function body is unchanged, reuse previous
                    analysis results
            else if is_scope_container(old_decl):
                // Recursive step for nested scopes
                diff_scopes(old_decl.scope, new_decl.scope, changes)

            // Remove matched declarations to identify new and removed
                ones
            remove identifier from old_declarations
            remove identifier from new_declarations

    // Identify newly introduced declarations
    for each identifier in new_declarations.keys():
        let new_decl = new_declarations[identifier]
        add_change(changes, "ADDED", new_decl)

    // Identify removed declarations
    for each identifier in old_declarations.keys():
```

```
        let old_decl = old_declarations[identifier]
        add_change(changes, "REMOVED", old_decl)

function calculate_scope_hash(scope):
    // 3. Pruning unchanged branches of the AST
    // Hash calculation is order-insensitive for non-function scopes
    // This can be achieved by either:
    // a) Sorting declarations by a stable order before hashing
    // b) Using a position-independent hash function
    if is_function_scope(scope):
        return hash_of_ordered_declarations(scope.declarations)
    else:
        return hash_of_order_independent_declarations(scope.
            declarations)

function create_map_of_declarations(scope):
    // Creates a map of declarations keyed by a unique signature (
        identifier + attributes)
    // This allows for efficient lookup of declarations
    let declaration_map = new Map()
    for each declaration in scope.declarations:
        let signature = generate_signature(declaration)
        declaration_map[signature] = declaration
    return declaration_map

function has_function_body_changed(old_function, new_function):
    // Compares the hash of function bodies to efficiently check for
        changes
    // Hashes are pre-computed during parsing
    return old_function.body_hash != new_function.body_hash

function is_function(declaration):
    // Checks if a declaration is a function
    return declaration.type == "Function"

function is_scope_container(declaration):
    // Checks if a declaration contains a nested scope
    return declaration has inner_scope

function add_change(changes, change_type, declaration):
    // Adds a record of the change to the list of changes
    changes.append({type: change_type, declaration: declaration})
```

Listing 2: Pseudo-code for the hierarchical scope-based diffing algorithm