Programming Language Tooling for Hylo
Incremental Compilation and Analysis

Author: Dobrin Bashev
d.bashev-1@student.tudelft.nl
Responsible professor: Andreea Costea
Supervisor: Jaro Reinders

1. Introduction

Fast, responsive tooling is essential for modern software development. Incremental compilation[1] – recompiling only modified code – helps reduce build times and improve developer productivity. While languages like Rust and Swift support this well, **Hylo**, a newer systems language focused on performance and safety, currently lacks incremental compilation support.
This project explores how to adapt incremental compilation techniques to Hylo's unique semantics and compiler architecture.

Research Questions:
RQ1: What incremental compilation techniques are used in modern languages?
RQ2: What are Hylo's key language features and compiler architecture?
RQ3: Which techniques can be adapted for Hylo, and how can they be conceptually prototyped?

2. Methodology

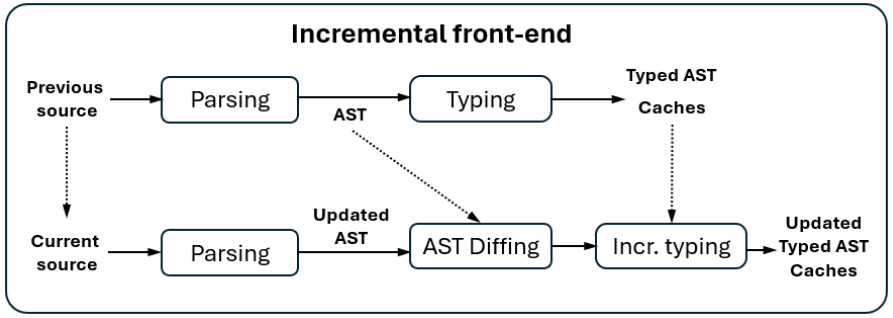
This research explored how incremental compilation could be integrated into the Hylo programming language using a mixed-method approach:

1) Literature review: Surveyed academic papers and technical documentation of compilers to identify and compare existing approaches to incremental compilation
2) Code review and expert consultation: Studied Hylo's compiler front-end and consulted with core developers to understand internal design and constraints.
3) Exploratory experiments: Conducted controlled experiments on synthetic Hylo code to assess performance bottlenecks and guide conceptual design.

A conceptual model was developed early in the project to guide feasibility analysis. Experiments showed that typing is the main bottleneck, making it the primary target for incrementalization.

The model proposes:

- AST diffing [2]: Track and classify edits between code versions at the syntax tree level, enabling precise change detection.
- Incremental type checking: Recheck only affected code based on AST changes and dependency tracking, avoiding full re-analysis after each edit.

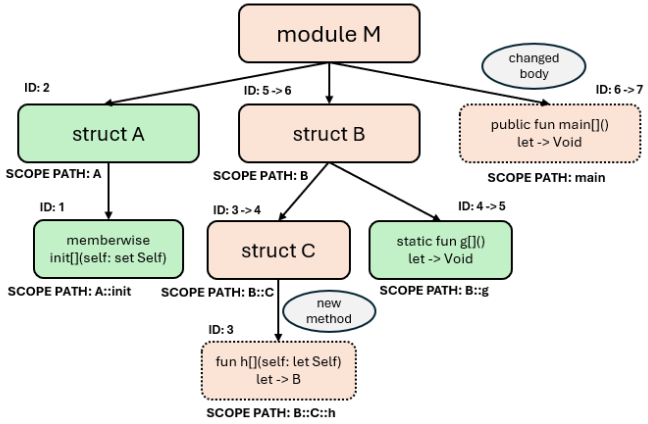


3. Incremental compilation in Hylo

Classification of program changes

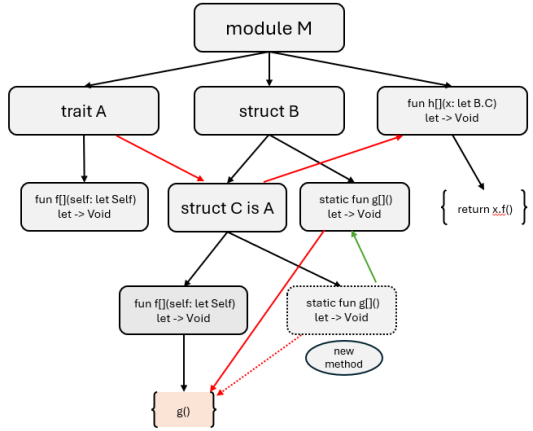
- Changes to function or method bodies without modifying signatures are local and can be re-type-checked in isolation
- Reordering declarations in scopes where order is semantically irrelevant has no impact and can be safely ignored
- Renaming is semantically meaningful and must be treated as a removal and insertion unless all references are updated atomically
- Modifying a declaration's API affects dependent scopes; conservative or fine-grained dependency graph invalidation is required

Hierarchical scope-based diffing algorithm



A hierarchical scope-based diffing algorithm matches declarations by stable identifiers, maps AST node IDs across versions, and uses scope-aware hashing to detect and isolate meaningful semantic changes.

Dependency tracking



During type checking, Hylo tracks dependencies by recording name resolutions, enabling selective invalidation of affected methods when changes shadow or expose declarations in scope.

4. Discussion and future work

The proposed incremental compilation model for Hylo draws on insights from both Rust and Swift. Like Rust, it avoids incremental parsing due to limited performance benefits and high complexity. For semantic analysis, it takes inspiration from Swift, using explicit dependency tracking instead of Rust's query-based model, reducing indirection while still enabling efficient incremental updates.

Currently, this model is theoretical. Future work will focus on:

- Implementing the approach in the Hylo compiler,
- Measuring real-world performance improvements, and
- Iteratively refining the model based on implementation feedback.

5. 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), 2020.
[2] 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.