DEBUGGING HYLO

PROVIDING DEBUGGING SUPPORT TO A MODERN, NATIVELY-COMPILED PROGRAMMING LANGUAGE

TUDelft

01. INTRODUCTION

→ Developers spend half of their time debugging software[1]. **Debuggers** are powerful tools that aid us in this process, providing features like **breakpoints**, **variable inspection**, and **line stepping.**
→ While debugging is seamless in mature languages, adding it to emerging ones like Hylo[2] presents challenges. Debugging support is crucial for improving usability and driving language adoption.
→ Research question: How can modern debugging infrastructure be used to support source-level debugging of Hylo code?

→ **Debuggers are complex**, relying on the Operating System and CPU for features such as breakpoints.
→ **Issue 1:** Debugging is platform-dependent; **Solution**: Debuggers like LLDB[3] bridge the gap across platforms.
→ **Issue 2**: Assembly-level debugging is impractical; **Solution:** The compiler bridges this gap by emitting **debug information.**
→ Debug information allows the debugger to reconstruct source-level details (e.g., variables). Compilers typically emit this information in a standardised format (e.g., DWARF[4]).

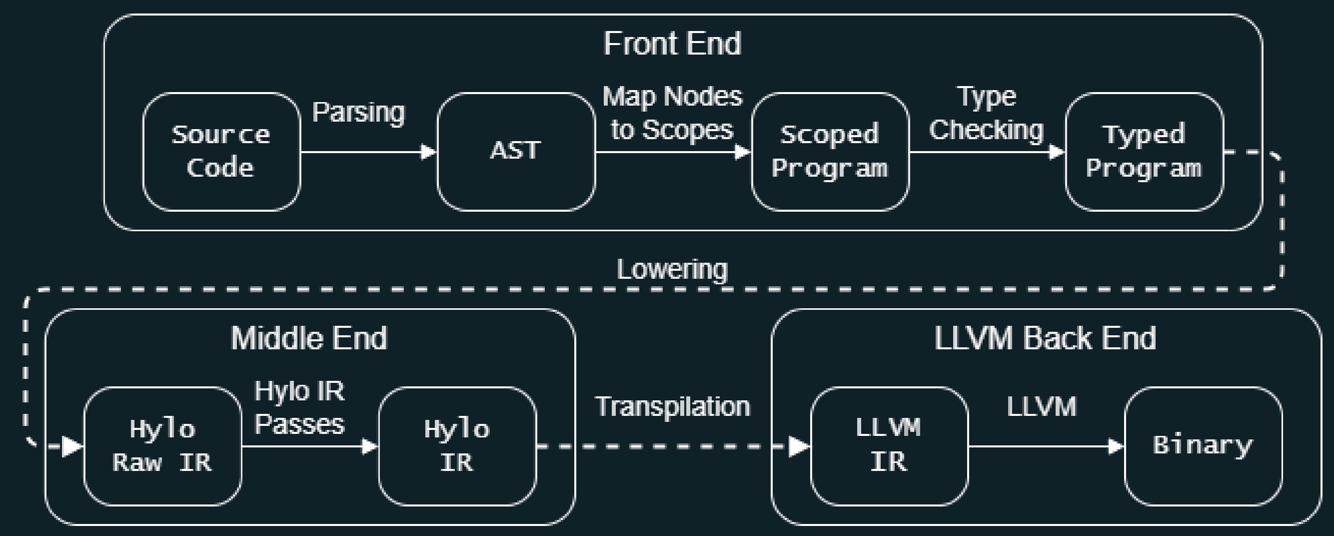

Figure 1: The architecture of the Hylo Compiler

→ The Hylo Compiler is LLVM-based[5] (i.e., the Hylo Compiler translates the Hylo code to LLVM's internal representation (LLVM IR), which is then compiled by LLVM to machine code).
→ If source-level information is passed to LLVM IR, LLVM can generate the DWARF information for our platform.
→ The Hylo IR already preserves some source-level information (e.g., for accurate error diagnostics).

02. METHODOLOGY

→ **Strategy**: Enhanced the Hylo Compiler to emit DWARF information. LLDB parses this metadata and enables source-level debugging.
→ **Observation-Driven Approach**: Studied Clang's[6] DWARF output for C++ and replicated it for similar Hylo constructs.
→ **Incremental Design**: Gradually added DWARF support for Hylo constructs, enabling one core debugging feature at a time.
→ **Prototype Hylo Compiler**: Extended Hylo's Compiler to implement our design, showcasing its practicality.

03. DESIGN

SOURCE LISTING

→ **LLDB Command**:
`source list -n <function-name>`
→ **Requirement**: Encode each Hylo function definition type (i.e., global, member, generic) to DWARF information.
→ **Approach**: Modify the compiler's transpilation phase to propagate function definition information from Hylo IR to LLVM IR.

```
(lldb) source list -n add
File: /workspaces/example.hylo
 32
 33   fun add(v: Vector2, w: Vector2) -> Vector2 {
 34       let x = v.x + w.x
 35       let y = v.y + w.y
 36       return Vector2(x: x, y: y)
 37   }
```

Figure 2: Sample LLDB output of the source list command

BREAKPOINTS + LINE STEPPING

→ **LLDB Commands**:
`break set -n <function-name>, step, next, finish`
→ **Requirement**: Annotate LLVM IR instructions with metadata that associates them with their corresponding locations in the source code.
→ **Approach**: Modify the compiler's transpilation phase to propagate source-level metadata from each Hylo IR instruction to its corresponding LLVM IR instruction(s). This approach works in most cases, with a few exceptions.

```
   (lldb) break set -n add
   (lldb) run
   * thread #1, name = 'testprogram', stop reason = breakpoint 1.1
   32
   33   fun add(v: Vector2, w: Vector2) -> Vector2 {
-> 34       let x = v.x + w.x
   35       let y = v.y + w.y
   36       return Vector2(x: x, y: y)
   37   }
```

Figure 3: Sample LLDB output of the breakpoint command

VARIABLE INSPECTION

→ **LLDB Command**:
`print <variable-name>; frame variable`
→ **Requirement**: Encode Hylo's type system and variables to DWARF.
→ **Approach:** We modify the transpilation phase to encode types, local variables and function parameters. For user-defined structures, we extend the lowering phase to recover necessary information. Additionally, we adjust the LLVM IR emission for function parameters.

```
(lldb) frame variable
(const Vector2 &) v = 0x00007fffffffddd8: {
  x = (value = 1)
  y = (value = 2)
}
(const Vector2 &) w = 0x00007fffffffdde8: {
  x = (value = 1)
  y = (value = 2)
}
(const Int) x = (value = 2)
(const Int) y = (value = 4)
```

Figure 4: Sample LLDB output of the frame variable command, executed after the breakpoint shown in Figure 3.

04. LIMITATIONS & FUTURE WORK

→ We emit accurate DWARF info for key Hylo features, such as variables, functions, user-defined types and generics.
→ We identified and explored the following limitations:

1. **Scope Modelling**: We assume that variables live throughout the function body, ignoring nested scopes (e.g., if statements), or Hylo's fine-grained variable lifetimes.
2. **Expression Evaluation**: LLDB relies on Clang, limiting expression evaluation support. The Hylo compiler could be integrated via an LLDB plugin.
3. **Existential Types**: Hylo has dynamic types (i.e., existentials). An LLDB plugin is required to show the concrete runtime types of existentials.

→ Future work may include IDE integration and debugging Hylo's concurrency model.

REFERENCES

[1] Abdulaziz Alaboudi and Thomas D LaToza. "An exploratory study of debugging episodes". In: arXiv preprint arXiv:2105.02162 (2021).
[2] Dimitri Racordon et al. "Implementation Strategies for Mutable Value Semantics". In: Journal of Object Technology 21.2 (2022). doi: 10.5381/jot.2022.21.2.a2. url: https://www.jot.fm/issues/issue_2022_02/article2.pdf.
[3] https://lldb.llvm.org/
[4] https://dwarfstd.org/doc/DWARF5.pdf
[5] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: International symposium on code generation and optimization, 2004. CGO 2004. IEEE. 2004, pp. 75–86.
[6] https://clang.llvm.org/

Student: Tudor-Stefan Magirescu
Contact: magirescu@student.tudelft.nl

Professor: Andreea Costea
Supervisor: Jaro Reinders