Debugging Hylo

Providing Debugging Support to a Modern, Natively-Compiled Programming Language

Tudor-Stefan Magirescu
Supervisor(s): Andreea Costea, Jaro Reinders
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: Tudor-Stefan Magirescu
Final project course: CSE3000 Research Project
Thesis committee: Andreea Costea, Jaro Reinders, Harm Griffioen

Abstract

Debugging is a fundamental part of software development, yet adding debugger support to new programming languages remains a complex and underexplored challenge. This report presents the design and implementation of source-level debugging for Hylo, a new systems programming language that emphasises generic programming and mutable value semantics. To achieve this, we extend the Hylo compiler to emit DWARF debug information, enabling seamless integration with the LLDB debugger. Our approach is incremental and guided by the behaviour of the Clang C++ compiler, allowing us to support Hylo core language features, such as variables, functions, user-defined types, and generics. We also identify and analyse key limitations, including challenges in representing Hylo's fine-grained variable lifetimes and projections, enabling evaluation of complex expressions in LLDB, and supporting Hylo's dynamic types (i.e., existentials). Beyond Hylo, this work aims to outline a general, reusable methodology for equipping new programming languages with modern debugging capabilities, improving their usability and adoption.

1 Introduction

Debugging is a fundamental yet time-consuming part of a software engineer's workflow, with studies indicating that developers spend roughly half of their programming time debugging software [1]. Given this importance, the development of reliable debugging tools is crucial to enhance developer productivity and ensure software correctness.

While modern, production-ready programming languages typically offer robust debugging support, extending similar capabilities to new languages remains a complex task. This process demands a deep understanding of systems programming, compiler internals, language semantics, and the debugging infrastructure. Prior work has explored the debugger internals [2, 3] and documented standards for debugging metadata [4]. Additionally, frameworks such as LLVM [5] facilitate the generation of debug information in a platform-independent manner. Nevertheless, there remains a lack of practical guidance on using these tools to implement effective debugging support for emerging languages.

Our work explores the process of adding source-level debugging to **Hylo**, a new systems programming language that emphasises generic programming [6] and a novel discipline called mutable value semantics [7]. As Hylo moves towards production readiness, the absence of debugging infrastructure presents a significant barrier to adoption and practical use. As such, we investigate the following research question: **How can modern debugging infrastructure be used to support source-level debugging of Hylo code?**

To address this question, this technical report makes the following contributions:

- The first systematic study of Hylo source-level constructs (e.g., variables, functions, types) and how they can be encoded into debug information. (**Section 4**)
- We identify challenging debugging and Hylo language features and suggest ways to support them in the future. (**Section 5**)
- The first Hylo compiler capable of emitting accurate debug information, enabling meaningful source-level debugging for a significant subset of the language.

Our approach is to extend the Hylo compiler to emit DWARF debug information [4], which can be interpreted by the LLDB debugger [8]. Using insights from the Clang compiler [9], we incrementally enhance Hylo's compilation pipeline to emit accurate metadata for a significant language subset. **Section 4** outlines this design, while **Section 5** discusses our approach's limitations. In particular, we describe how complex variable lifetimes (**Section 5.1**) and existential types (**Section 5.3**) pose a challenge for the debugging process. Furthermore, we discuss the limitations of relying on Clang for expression evaluation in LLDB and describe how the Hylo compiler could be used instead (**Section 5.2**).

2 Background

Although debuggers come in a variety of flavours, we can describe their core capabilities in an abstract manner. The debugger attaches to the debugee (i.e., the process one wants to debug) and controls its execution. For instance, the debugger can start or stop the execution of the debugee or inspect memory addresses while the debugee is running. More specifically, most debuggers provide the following basic features: **breakpoints**, which halt the execution of the debugee at a specific location; **line/instruction stepping**, which advance the execution of the debugee with one source code line/instruction; and **memory/variable** inspection, allowing the debugger to read the value present at a specific memory address or the value of a variable. Most often, the user (i.e., the programmer) interacts with the debugger through a command-line interface (**CLI**) or an integrated development environment (**IDE**).

2.1 The Anatomy of a Debugger

Three components form the essential pillars that sustain the debugger's functionality: the **operating system**, the **hardware** and the **compiler**. The first is the operating system, which provides a debugging API. For example, the Linux operating system [10] exposes a series of system calls (**syscalls**) that allow the debugger to control the execution of the debugee. More specifically, syscalls such as `fork` [11], `exec` [12], and `waitpid` [13] are used by the debugger to run the debugee and to regain control when the debugee halts (e.g. when it hits a breakpoint). Perhaps the most useful syscall is `ptrace` [14], which is crucial for handling the core debugger features outlined above. By using `ptrace`, the debugger can: attach to the debugee, read/write register values set by the debugee, execute the debugee's next instruction, etc. In contrast, the Windows operating system debugging API is higher-level and more straightforward, though this comes at the cost of reduced granularity [2].

Generally, the operating system API provides user-space programs with an abstraction over the underlying **hardware**. However, this is not the case for debuggers, which need to be aware of the CPU's instruction set architecture (**ISA**). For example, reading a register value with `ptrace` requires passing a pointer to a platform-specific struct, which holds ISA-defined register fields. Additionally, CPUs offer specific support for some of the debugging features. For instance, the x64 Intel ISA provides the special `int3` instruction, which causes a software interrupt when executed [15]. The debugger inserts this instruction in the debugee's code when the user requests to set a breakpoint. Sy Brand [3] provides a detailed design of a debugger for the x64 ISA.

Using the concepts described above, we can already build a debugger targeting a specific operating system and ISA. In addition to being platform dependent, such a debugger would only support **machine code**-based debugging, such as setting breakpoints on specific instructions or reading CPU registers. In contrast, users typically expect to debug code written in a high-level language (i.e., set breakpoints on source lines, read variable values). To enable these features, debuggers require support from the high-level language's **compiler**, which constitutes the third pillar of a debugger. A compiler's primary role is to translate the source code written in a high-level language into machine code. During this process, compilers can choose to preserve certain metadata, called **debugging information**, which they make available to the debugger. This metadata should provide enough context to enable the debugger to operate at the source language's level.

2.2 The DWARF Debug Information Format

Oftentimes, compilers emit debugging information that adheres to a well-known format. **DWARF** is one such open-source debugging information format that aims to be language and platform-independent [4]. DWARF information is structured in multiple sections, each aiding the debugger in reconstructing details about the source code.

One of these sections is `debug_info`, which contains the list of debugging information entries (**DIEs**). These DIEs provide details regarding the source code symbols (e.g., variables, functions, parameters). In particular, a language's **type system** can be encoded in the DIEs, allowing a debugger to infer how to present the data to the user. Additionally, DWARF assumes a language's symbols are tied to specific **lexical scopes**. To encode this scoping structure, the sequence of DIEs is represented as a tree, where the parent of a node indicates its scope. Another section of the DWARF information is the `debug_line`, which contains the **line table**, a data structure which maps machine-code instructions to their corresponding source code locations. As we shall see in Section 4.2, the line table is crucial, allowing a debugger to enable source-level breakpoints and line stepping. We will describe the various aspects of the DWARF format more thoroughly in Section 4, when we detail the required information to enable certain debugging features at the source code level.

2.3 The LLDB Debugger

From the previous discussion, one can notice that debuggers are inherently platform-dependent. Building a new debugger with support for multiple operating systems and ISAs is a monumental engineering task. Fortunately, there exist well-established debuggers that can bridge the gap between the various platforms.

One such debugger is LLDB [8], which is part of the LLVM [5] project. It was designed to provide a modern, high-performance debugging experience that aligns with LLVM's modular architecture and reusable component philosophy. Unlike traditional debuggers, LLDB is structured as a set of libraries, making it suitable not only for use via its command-line interface but also as a programmable backend through Python scripting or integration with other tools and IDEs. Although LLDB's architecture was originally designed to allow adding support for new programming languages [16], there is still an ongoing effort to transform LLDB into a more versatile "debugger toolkit" rather than one primarily tailored for Clang-based languages (e.g., C, C++, Objective-C) [17]. Nevertheless, several LLVM-based programming languages actively address these limitations to improve their debugging experience with LLDB [18].

2.4 The Hylo Programming Language

Hylo is a modern systems programming language that aims to balance efficiency, safety, and conceptual simplicity [19]. Hylo supports these goals with its strong emphasis on generic programming [6] and a novel discipline called mutable value semantics [7].

Put simply, mutable value semantics imply that references are treated as second-class citizens in Hylo. This means the programmer cannot explicitly manipulate or store references, since reference types are not part of Hylo's type system. Consequently, variables in Hylo do not share mutable state, eliminating a broad class of bugs related to aliasing and unintended side effects.

Hylo is an ahead-of-time, natively compiled programming language. Hylo's compiler can be split into three components: a front end, a middle end and a back end. Hylo's **front end** performs lexing, parsing, type checking, and lowering to Hylo intermediate representation (**IR**). The **middle end** performs checks and optimisations on the Hylo IR. Lastly, Hylo's backend transpiles Hylo IR to LLVM IR, which is then compiled to machine code by LLVM [5]. As such, Hylo is an LLVM-based language, leveraging LLVM's mature infrastructure and optimisation capabilities. Figure 1 illustrates the compiler's architecture.

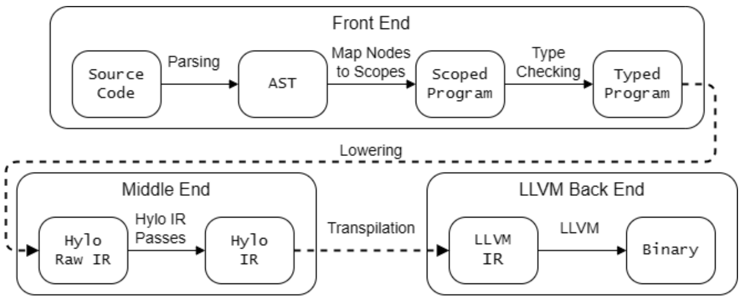


Figure 1: Architecture of the Hylo compiler.

Currently, Hylo offers limited tooling support. Notably, its compiler does not emit debugging information, which prevents Hylo programs from being debugged. However, some source-level metadata is preserved through phases of the compilation pipeline, up to the Hylo IR (e.g., to report accurate error diagnostics). As we will see throughout this report, enabling debugging support primarily involves modifying the **transpilation** phase of the compiler, where Hylo IR is translated to LLVM IR. Specifically, we use LLVM's debug API [20] to generate the necessary DWARF information.

3 Methodology

This report presents the design and implementation of source-level debugging support for the Hylo programming language. Our approach is to modify the Hylo compiler to emit DWARF metadata, allowing Hylo programs to be debugged using LLDB. We selected the DWARF standard because it is open-source [4], platform and language-independent, and well-supported within the LLVM ecosystem. Likewise, we chose LLDB for its tight integration with LLVM, modern architecture, and its growing support for source-level debugging across various programming languages [18].

In Section 4, we explore a few fundamental debugging features: **source listing**, **breakpoints and line stepping** and **variable inspection**. Our decision to focus on these features is not based on their commonality or usefulness, although both might be true. Instead, these features lie at the foundation of a debugger, meaning they serve as building blocks to enable more complex debugging features. Additionally, we show that providing debugging support to a programming language can be incremental, as each core feature allows us to focus on a different subset of DWARF information. Our ordering of these features reflects their complexity and the required engineering work.

We add source-level debugging support to Hylo by studying how Clang emits DWARF information for C++, one of the few languages with built-in support in LLDB. By representing Hylo constructs in DWARF similarly to how Clang encodes analogous C++ constructs, we allow LLDB to work out-of-the-box for a substantial subset of the language. This approach avoids the considerable effort of developing and maintaining Hylo-specific LLDB plugins or a custom fork, and helps us precisely identify where this deeper integration with LLDB is required. We discuss such scenarios in Sections 5.2 and 5.3, allowing the Hylo team to assess whether the additional engineering effort is worthwhile.

Throughout the report, we use the examples shown in Figure 2 to illustrate our design choices. The code snippets showcase the same 2D vector translation algorithm written in both C++ and Hylo. The Hylo example highlights the language features for which we emit accurate debug information, including **variables**, **functions**, **built-in** and **generic types**, **user-defined types** and **member functions**. Unless otherwise specified, we imply that the C++ example is compiled using Clang with full debug information (**-g**) and no optimisations (**-O0**) and that the Hylo code is compiled without any LLVM optimisations.

```cpp
#include <iostream>

template <class T>
T add(T n, T m) {
    return n + m;
}

struct Vector2 {
    int x;
    int y;

    void offset(const Vector2& delta) {
        x = add(x, delta.x);
        y = add(y, delta.y);
    }
};

int main() {
    Vector2 v{1, 1}, delta{1, 1};
    v.offset(delta);
    std::cout << v.x << '\n';
}
```

(a) C++ example

```
fun add<T: AdditiveArithmetic>(_ n: T, _ m: T) -> T {
    return n + m
}

type Vector2: Deinitializable {
    public var x: Int32
    public var y: Int32
    public memberwise init

    public fun offset(_ delta: let Vector2) inout {
        &x = add(x, delta.x)
        &y = add(y, delta.y)
    }
}

public fun main() {
    var v = Vector2(x: 1, y: 1)
    let delta = Vector2(x: 1, y: 1)
    &v.offset(delta)
    print(v.x)
}
```

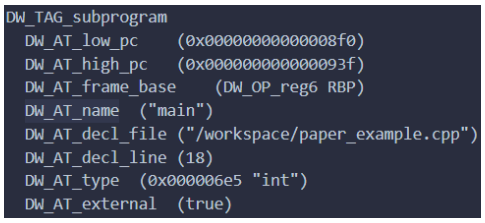
(b) Hylo example

Figure 2: Vector translation example in both: a) C++ and b) Hylo

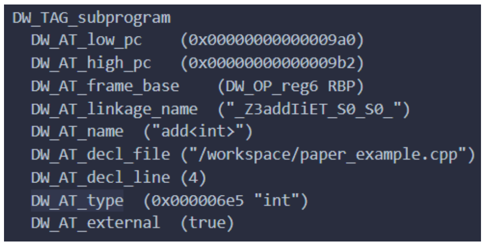
Beyond design, we implement these ideas in a fork of the Hylo compiler. This represents the first Hylo compiler capable of emitting accurate DWARF debugging information, a milestone that marks a significant step toward making Hylo a production-ready language.

4 Design of Debugging Support for Hylo

This section outlines our design for source-level debugging in Hylo. Each subsection focuses on a core debugging feature, detailing the required DWARF metadata and how LLDB uses it. We examine how Clang emits this information for C++ and describe the corresponding modifications to the Hylo compiler.



```
DW_TAG_subprogram
  DW_AT_low_pc    (0x00000000000008f0)
  DW_AT_high_pc   (0x000000000000093f)
  DW_AT_frame_base    (DW_OP_reg6 RBP)
  DW_AT_name  ("main")
  DW_AT_decl_file ("/workspace/paper_example.cpp")
  DW_AT_decl_line (18)
  DW_AT_type  (0x000006e5 "int")
  DW_AT_external  (true)
```



```
DW_TAG_subprogram
  DW_AT_low_pc    (0x00000000000009a0)
  DW_AT_high_pc   (0x00000000000009b2)
  DW_AT_frame_base    (DW_OP_reg6 RBP)
  DW_AT_linkage_name  ("_Z3addIiET_S0_S0_")
  DW_AT_name  ("add<int>")
  DW_AT_decl_file ("/workspace/paper_example.cpp")
  DW_AT_decl_line (4)
  DW_AT_type  (0x000006e5 "int")
  DW_AT_external  (true)
```

Figure 3: The Subprogram DIEs corresponding to the `main` (on the left) and `add` (on the right) functions defined in the C++ code in Figure 2.

4.1 Source Listing

The **source list** command in LLDB is a powerful tool for viewing code directly within the debugger. With a search query, users can display relevant sections of the codebase. This command is invaluable not only for navigating code during a debugging session but also as a preliminary step before setting a breakpoint. This section analyses the scenario in which the source code is listed by a given **function name**. The specific LLDB command is: `source list -n <function_name>`.

When available, LLDB uses the DWARF information to search for a function given its name. More specifically, LLDB searches through the `debug_info` section of the metadata, which contains the tree of debug information entries (**DIEs**). In particular, LLDB parses the `DW_TAG_subprogram` DIEs, which contain information regarding the functions defined in the source code.

Figure 3 illustrates the structure of a `DW_TAG_subprogram` DIE. As we'll also see in the following sections, each DIE consists of a list of attributes. Common ones like `DW_AT_name`, `DW_AT_decl_file`, and `DW_AT_decl_line` appear across many DIE types and capture source-level details, such as the symbol's name (in this case, the function name) and its location in the source code. The attribute `DW_AT_linkage_name` is specific to subprogram DIEs, and encodes the function's actual name in the binary. Since languages like C++ and Hylo support function overloading, compilers use **name mangling** to avoid symbol conflicts in the compiled machine code. Most other attributes (e.g., `DW_AT_low_pc`) contain machine-level details about the function and are synthesised by the LLVM debug API, meaning they do not concern our implementation.

We briefly discuss how the value of the `DW_AT_name` attribute varies with the type of function. Figure 3 shows examples of global (e.g., `main`) and template (e.g., `add`) functions. Clang records global function names as they appear in the source code, while template function names include their template arguments (e.g., `add<int>`). This is because C++ template functions are **monomorphised** (i.e., a specialised instance for each set of type arguments is compiled [21]), so encoding the template arguments in `DW_AT_name` helps LLDB distinguish between specialisations. Member functions (e.g., `offset`) are also recorded as they appear in the source code. Them being member functions is indicated by their parent DIE being a structure type DIE (see Section 4.3.1 for details on types).

In Hylo, each non-generic function (e.g., `main`, `offset`) is compiled into a single Hylo IR function. Member functions are compiled similarly to global functions, with an additional parameter for the `self` pointer. Generic functions (e.g., `add<T>`) are monomorphized in a dedicated Hylo IR pass, meaning a separate Hylo IR function is created for each specialisation. Afterwards, Hylo IR functions are translated one-to-one to LLVM IR functions.

To enable source listing for Hylo code, we use the LLVM debug API to annotate each LLVM IR function definition with debug metadata during the compiler's **transpilation** phase. To generate the subprogram DIE, LLVM requires details such as the function name, declaration file and line, and mangled name. We propagate the first three from Hylo IR and compute the latter using Hylo's mangling algorithm. We populate the `DW_AT_name` attribute following the previously discussed Clang conventions.

4.2 Breakpoints and Line Stepping

Breakpoints and line stepping are fundamental to the debugging workflow. Breakpoints pause the execution of the debugee at a specified point, allowing the user to inspect or alter the program state. Execution can then be resumed incrementally, using one of the flavours of stepping: **step into**, **step over**, **step out**. Without debug information, these features operate at the **instruction** level, which is undesirable for high-level code. With debug info, users can interact at the **source line level** instead, by setting breakpoints on specific lines or stepping through lines of code.

To support this higher-level view, LLDB uses the `debug_line` section of DWARF, which contains the **line table**, a mapping from machine instructions to their corresponding source locations. If the compiler generates an accurate DWARF line table, LLDB can offer source-level stepping and breakpoints, by using similar algorithms to those covered in Chapter 14 of Sy Brand's work [3].

LLVM can generate a line table if the LLVM IR instructions are annotated with source-level metadata. This metadata includes the original line number, column number, and lexical scope of the corresponding Hylo source code. In our work, we assume the scope of a source code line is the enclosing `DW_AT_subprogram` DIE representing the function it belongs to, a limitation we discuss in Section 5.1. Since Hylo IR already carries source metadata, we update the compiler's **transpilation phase** to propagate it to LLVM IR.

Figure 4 illustrates the similarities between Hylo IR and LLVM IR. Both are linear, static single-assignment (**SSA**) intermediate representations. The fundamental unit of execution is a **basic block**, a sequence of instructions with no internal control flow (e.g., `b0` in Figure 4a and `prologue` in Figure 4b). Functions in both IRs consist of basic blocks forming a **control flow graph**. During the **transpilation phase**, each Hylo IR instruction is translated into zero or more LLVM IR instructions. Basic blocks are mapped one-to-one, except that stack allocations are moved to a dedicated `prologue` block.

As such, we identify a general heuristic for metadata propagation: we copy source-level information from each Hylo IR instruction to its corresponding LLVM IR instructions. While effective in most cases, this causes misleading debugger behaviour in two scenarios, which we cover below: **stack allocations** and **default argument construction**.

As previously described, stack allocations in Hylo IR are relocated to a dedicated `prologue` block in LLVM IR. Each LLVM IR `alloca` instruction corresponds to a Hylo IR `alloca_stack` instruction. However, the latter might originate from non-consecutive lines in the source code. Applying our heuristic in this case causes LLDB to display non-linear stepping behaviour, appearing to jump unpredictably across the function. To address this, we omit source-level annotations in the `prologue` block. This ensures that when LLDB enters a function (e.g., by hitting a function-level breakpoint), it executes the prologue, halting only at the first instruction with source-level metadata.

Hylo supports default function arguments. For example, while only one argument is explicitly passed to the `print` function called in Figure 2b (i.e., `v.x`), it accepts two **optional**

```
external fun main() -> {} {
b0(%b0#0 : &{}):
  %i0.0: &Vector2 = alloc_stack Vector2
  %i0.2: &word = subfield_view %i0.0, 0, 0
  %i0.3: &word = access [set] %i0.2
  store i64(0x1), %i0.3
  end_access %i0.3
  %i0.6: &word = subfield_view %i0.0, 1, 0
  %i0.7: &word = access [set] %i0.6
  store i64(0x1), %i0.7
  end_access %i0.7
```

(a) Hylo IR

```
define private void @main(ptr noalias nocapture nofree %0) {
prologue:
  %4 = alloca %Vector2, align 8
  br label %b0

b0:                                       ; preds = %prologue
  %5 = getelementptr %Vector2, ptr %4, i32 0, i32 0, i32 0
  store i64 1, ptr %5, align 8
  %6 = getelementptr %Vector2, ptr %4, i32 0, i32 1, i32 0
  store i64 1, ptr %6, align 8
```

(b) LLVM IR

Figure 4: A section of the `main` function from Figure 2b compiled to Hylo IR (on the left) and LLVM IR (on the right). More specifically, this section corresponds to the `Vector2` instantiation on line 17.

arguments, which are constructed in the caller's (i.e., `main`'s) stack frame. However, the corresponding Hylo IR instructions carry source-level metadata pointing to the source location of `print`'s definition (i.e., `Print.hylo`). To avoid this issue, we annotate the LLVM IR instructions that create default arguments with the same metadata as the function call itself, ensuring they appear as part of the same source line to the debugger.

4.3 Variable Inspection

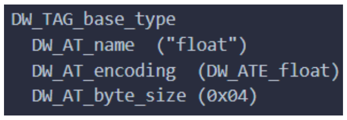
Another core capability of debuggers is their ability to inspect the debugee's memory during execution. Without debug information, the debugger can only inspect registers or memory addresses. Instead, users would prefer to inspect high-level constructs, such as variables. If debug information is available, LLDB enables variable inspection via commands such as `print <variable-name>`.

This section presents the necessary modifications to the Hylo compiler to enable variable inspection. First, we explore how Hylo's type system can be encoded in DWARF (Section 4.3.1). Second, we use the type information to encode individual variables, namely function parameters and local variables (Section 4.3.2).

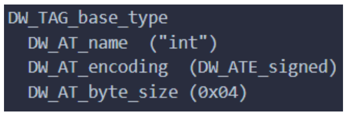
4.3.1 DWARF Type DIEs

Type information is crucial for debuggers like LLDB because it describes how variables are laid out in memory, enabling raw memory to be interpreted and displayed as readable variable values. In DWARF, types are represented through DIEs, which are flexible enough to encode a broad range of type systems. DWARF includes built-in support for common **fundamental types** (e.g., `int`, `float`, `char`) and mechanisms to construct more complex types (e.g., structs, arrays, classes). To illustrate how Hylo's types can be mapped to DWARF DIEs, we focus on **built-in types** and **user-defined structures** (e.g., the `Vector2` type shown in Figure 2b). In our version of the Hylo compiler, the emission of DWARF type information is handled by a dedicated subroutine which is invoked whenever the type of a variable must be described. This subroutine determines the corresponding Hylo type and constructs the appropriate DIE.

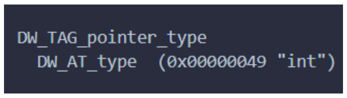
Figure 5 shows how the C++ types `int` and `float` are represented in DWARF. Although both DIEs have tags `DW_TAG_base_type`, we differentiate them by the `DW_AT_encoding` attribute: `DW_ATE_float` denotes a floating-point number, while `DW_ATE_signed` represents



```
DW_TAG_base_type
  DW_AT_name  ("float")
  DW_AT_encoding  (DW_ATE_float)
  DW_AT_byte_size (0x04)
```



```
DW_TAG_base_type
  DW_AT_name  ("int")
  DW_AT_encoding  (DW_ATE_signed)
  DW_AT_byte_size (0x04)
```



```
DW_TAG_pointer_type
  DW_AT_type  (0x00000049 "int")
```

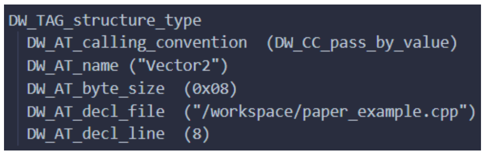
Figure 5: Type DIEs corresponding to a floating-point number (on the left), a four-byte signed integer (in the middle), and a pointer to an integer type (on the right).

a signed integer. Additionally, the `DW_AT_byte_size` attribute records the type's size in bytes, and `DW_AT_name` specifies its name.

DWARF also provides DIEs for encoding C++'s **compound types**, such as **pointers**, **references** or **const**-qualified types. For example, Figure 5 also displays the encoding of an `int*`. We notice it consists of a `DW_TAG_pointer_type` DIE, which includes a `DW_AT_type` attribute pointing to the `int` base type. More complex types, such as `const Vector&&`, are expressed by chaining multiple DIEs, with each representing a layer of the compound type.

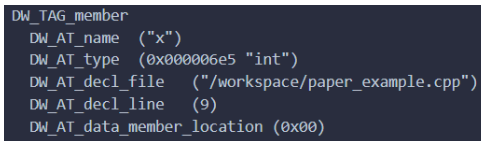
Hylo exposes three main classes of primitive types: **integers** (e.g., `Builtin.i32`), **floating-point numbers** (e.g., `Builtin.float32`), and **opaque pointers** (i.e., `Builtin.ptr`). Hylo defines built-in operations for these primitives (e.g., `Builtin.add_i32` for integer addition). Note that these primitives and operations are available only within Hylo's Standard Library. Instead, the programmer should use types such as `Int32` referenced in Figure 2, which are structures defined in the Standard Library that serve as abstractions over built-in types.

In our implementation, we encode Hylo's primitive types similarly to the examples shown in Figure 5. For integers and floats, we infer the size from the type name (e.g., `Builtin.i64` implies 8 bytes) and assign the corresponding `DW_AT_encoding`. Since Hylo's pointers are opaque (i.e., they hide the type of the pointer), we encode them as integers. Their `DW_AT_byte_size` is platform-dependent, but we retrieve it using the LLVM API, which exposes platform-specific details.



```
DW_TAG_structure_type
  DW_AT_calling_convention  (DW_CC_pass_by_value)
  DW_AT_name ("Vector2")
  DW_AT_byte_size  (0x08)
  DW_AT_decl_file  ("/workspace/paper_example.cpp")
  DW_AT_decl_line  (8)
```

(a) Top-level struct DIE



```
DW_TAG_member
  DW_AT_name  ("x")
  DW_AT_type  (0x000006e5 "int")
  DW_AT_decl_file   ("/workspace/paper_example.cpp")
  DW_AT_decl_line   (9)
  DW_AT_data_member_location (0x00)
```

(b) Member variable DIE

Figure 6: Encoding of the `Vector2` type and its member `x` from Figure 2.

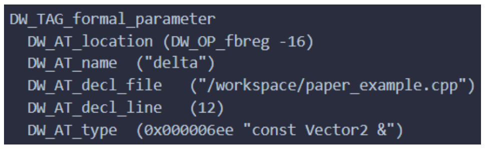
Figure 6 illustrates how a C++ user-defined structure like `Vector2` is encoded using DWARF DIEs. The encoding is split into two parts: a top-level DIE representing the struct itself (Figure 6a) and a set of child DIEs, each corresponding to a member field (Figure 6b). Both of these types of DIEs provide source-level information, such as the symbol name (`DW_AT_name`) and source location (`DW_AT_decl_file` and `DW_AT_decl_line`).

These DIEs also encode the memory layout. The struct DIE includes a `DW_AT_byte_size` attribute for the struct's size. Each member DIE contains a `DW_AT_data_member_location` indicating the offset of the field within the struct. Usually, encoding these details is challenging, since the struct's layout is **platform-dependent** (e.g., see the System V ABI for AMD64 [22]). Fortunately, LLVM exposes target-specific size and layout information, allowing us to query a struct's layout directly from its LLVM IR representation. Lastly, each member DIE contains a `DW_AT_type` attribute pointing to the field's type.

In our implementation of the Hylo compiler, each time a new Hylo IR struct type is **transpiled** to an LLVM IR type, we generate its struct DIE. We achieve this by recursively constructing the DIEs for its member fields before creating the top-level struct DIE. We cache these DIEs once computed to improve efficiency and prevent redundant metadata generation. Importantly, this approach supports Hylo's generic struct types as well. Since generic types in Hylo are monomorphized similarly to generic functions, they are already specialised by the time we reach the LLVM IR level. This means that we don't need to adapt our DWARF emission process to support generic types.

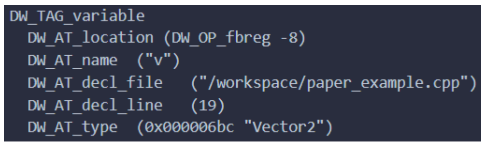
4.3.2 DWARF Variable DIEs

Having encoded types as DWARF DIEs, we now focus on representing Hylo source-code variables.



```
DW_TAG_formal_parameter
  DW_AT_location (DW_OP_fbreg -16)
  DW_AT_name  ("delta")
  DW_AT_decl_file   ("/workspace/paper_example.cpp")
  DW_AT_decl_line   (12)
  DW_AT_type  (0x000006ee "const Vector2 &")
```

(a) Function Parameter DIE



```
DW_TAG_variable
  DW_AT_location (DW_OP_fbreg -8)
  DW_AT_name  ("v")
  DW_AT_decl_file   ("/workspace/paper_example.cpp")
  DW_AT_decl_line   (19)
  DW_AT_type  (0x000006bc "Vector2")
```

(b) Local Variable DIE

Figure 7: The DWARF encoding of the `delta` function parameter and local variable `v` from the example in Figure 2a.

Figure 7 shows two kinds of variable DIEs: parameters (a) and local variables (b). Similar to before, these encode source-level information via the `DW_AT_name`, `DW_AT_decl_line`, and `DW_AT_decl_file` attributes. Additionally, each variable DIE includes a `DW_AT_type` attribute, the construction of which we described in Section 4.3.1. The variable's scope is determined by the DIE's parent. For now, we assume that a variable's scope is the scope of the function in which it is declared. We explore this limitation in Section 5.1.

A key attribute is `DW_AT_location`, which links variables to their memory locations. As previously discussed, type DIEs describe how to interpret a region of raw memory. Together with the location information, this enables LLDB to correctly display a variable's value. The DWARF standard [4] describes several methods for encoding variable locations. In the examples shown in Figure 7, both variables are stored on the stack, and their locations are represented as offsets from the stack base pointer.

Fortunately, since LLVM performs DIE generation, we don't need to handle the low-level details of the `DW_AT_location` attribute directly. Instead, we use LLVM's debug information API, which exposes **debug records** to track source-level variable values within LLVM IR [20]. Among the three types of debug records, we identify `dbg_declare` as the most suitable for use by compiler frontends, such as Clang or the Hylo compiler. In contrast, the remaining debug records are primarily intended for LLVM's internal use, enabling the preservation of variable information during optimisation passes [23].

Therefore, to represent source-level variables, the compiler should emit a `dbg_declare` record for each variable. This record includes metadata such as the variable's name, its DWARF type, and its source-code location. Crucially, it also contains a pointer to a **stable** memory address where the variable is allocated by the LLVM IR. In most cases, this is a pointer to the stack slot where the variable resides.

```
define void @offset(ptr %0, ptr %1) !dbg !885 {
  %3 = alloca ptr, align 8
  %4 = alloca ptr, align 8
  store ptr %0, ptr %3, align 8
    #dbg_declare(ptr %3, !886, !DIExpression(), !888)
  store ptr %1, ptr %4, align 8
    #dbg_declare(ptr %4, !889, !DIExpression(), !890)
```

(a) IR Generated by Clang

```
define private void @offset(ptr %0, ptr %1, ptr %2) !dbg !20 {
prologue:
  %debug.0 = alloca ptr, align 8
    #dbg_declare(ptr %debug.0, !21, !DIExpression(), !31)
  store ptr %0, ptr %debug.0, align 8
  %debug.1 = alloca ptr, align 8
    #dbg_declare(ptr %debug.1, !32, !DIExpression(), !31)
  store ptr %1, ptr %debug.1, align 8
```

(b) IR Generated by **our** Hylo Compiler

Figure 8: Examples of LLVM IR generated by both Clang and our version of the Hylo compiler for the `offset` function from Figure 2. The snippets have been simplified for clarity and only illustrate the stack allocations of the function parameters.

Let us first examine how parameter DIEs are emitted. Figure 8a shows part of the Clang-generated IR for the `offset` function. The function takes two pointers: the implicit `this` pointer and the `delta` argument (originally a `const Vector2&`). At the start of the function, Clang allocates two stack slots and stores the parameters there. Each slot is then referenced by a `dbg_declare`, along with type and source location metadata.

From this and similar examples, we infer that Clang lowers C++ pointer and reference parameter types to opaque pointers in LLVM IR, allocates them on the stack at function entry (we assume optimisations are disabled), and emits a corresponding `dbg_declare` tied to each stack slot.

A crucial insight is that Hylo's function parameters are compiled as **pass-by-reference**. While Hylo's type system doesn't expose reference types, the compiler lowers parameters to LLVM IR as opaque pointers, effectively treating them as references. This is consistent with one of Hylo's core principles: copies are **explicit by default** [24]. Correctness properties, such as the law of exclusivity[7], are enforced statically by the compiler before lowering to LLVM IR. Consequently, parameters can be safely passed by reference in the generated IR.

However, unlike Clang, Hylo does not emit redundant stack allocations for function parameters. Instead, parameters are used directly from the SSA registers. While this is efficient, it poses a challenge for debug info, as we require stable memory addresses to use `dbg_declare`. Our approach is to synthesise stack slots using `alloca` for each parameter at function entry, store the incoming value, and emit a corresponding `dbg_declare` referencing the allocated address, the parameter's type DIE, and source information. We apply this transformation to all but the last parameter, which is reserved for the function's return value. Figure 8b shows the result of our approach for the `offset` function. Note that in Hylo's member functions, the first parameter always represents the `self` object.

Although function parameters are compiled as pass-by-reference, Hylo defines several **parameter passing conventions** that carry semantic meaning describing how a parameter is passed to the callee. We aim to preserve this meaning in DWARF by mapping each Hylo convention to a C++ reference type with similar semantics (see Table 1a). These semantic similarities between Hylo's parameter conventions and C++'s reference types are explored in Hylo's official language tour [24]. We encode these reference types (i.e., compound types) following the same strategies discussed earlier in Section 4.3.1. Lastly, since LLVM IR uses opaque pointers, we recover the underlying type (e.g., `Vector2`) from the Hylo IR.

For local variables, we encode them as **value types** instead. The Hylo compiler allocates them on the stack with `alloca` LLVM IR instructions. We identify source-level variables by checking whether a given allocation corresponds to a user-defined variable (as opposed to, say, a constant). Hylo also defines several **bindings** for local variables (e.g., mutable/immutable), which we capture as shown in Table 1b. In short, variables introduced with `let` are treated as **const-qualified**, while others are not. A limitation of our approach occurs when variables share stack slots, which we discuss further in Section 5.1.

Table 1: Mappings between Hylo parameter passing conventions and local variable bindings to corresponding C++ constructs with similar semantics for the `Vector2` type.

(a) Parameter Convention Mappings

Hylo Convention	C++ Encoding
`let`	`const Vector2&`
`inout`	`Vector2&`
`sink`	`Vector2&&`
`set`	`Vector2&`

(b) Local Variable Binding Mappings

Hylo Binding	C++ Encoding
`let`	`const Vector2`
`var`	`Vector2`
`inout`	`Vector2`

5 Limitations of our Design

In the previous section, we detailed how we extended the Hylo compiler to emit DWARF information. The example in Figure 2 is representative of the features we emit accurate debug info for: variables, function types (global, member, generic), and both built-in and user-defined types, including generics. Though we focus on core functionality, our implementation already emits sufficient DWARF metadata to enable advanced debugging features, such as watchpoints, stack trace visualisation and variable assignments.

In this section, we outline key limitations of our current design and suggest directions for future improvements, focusing on improving local variable debugging (Section 5.1), Hylo expression evaluation in LLDB (Section 5.2), and support for existential types (Section 5.3).

5.1 Limitations in Variable Emission and Lifetime Tracking

In Section 4.3.2, we noted two limitations of our design: local variables may share memory, and variable scope is assumed to be function-level. We now discuss these in more detail.

When handling local variable DIEs, we rely on detecting stack allocations in LLVM IR that correspond to local variables. However, variables declared with `let` or `inout` bindings don't always create new stack allocations. In cases where they **project** existing values (e.g., `let a = b`, or `inout c = d.e`), the compiler may reuse the original memory instead, after statically enforcing code correctness. A potential improvement would be to detect these projections and introduce redundant stack allocations when possible, similar to our approach for parameters.

Additionally, we assumed that a variable's scope spans the entire function in which it is declared. However, this assumption fails when we declare variables inside nested scopes (e.g., within `if` statements). Instead, variable DIEs can be nested within `DW_TAG_lexical_block` DIEs, which indicate that their visibility is limited to a lexical scope.

This approach is generally sufficient for C++, where most local variables have automatic storage duration, meaning they are deallocated at the end of their lexical scope [21]. However, Hylo's variable lifetimes are more fine-grained. A local variable's lifetime begins after initialisation and ends either after consumption or after its last usage [25]. These lifetimes often do not align with lexical scopes. To more precisely reflect such semantics in

DWARF, one idea is to emit one artificial `DW_TAG_lexical_block` DIE for each variable, corresponding to the variable's actual lifetime rather than its lexical scope.

5.2 Unsafe Expression Evaluation

Section 4.3 discussed how we extended the Hylo compiler to emit variable DIEs, enabling variable inspection within LLDB. Beyond basic inspection, these DIEs enable **simple expression evaluation**, such as modifying variables at runtime, or performing arithmetic operations on Hylo variables of built-in types (e.g., `int`, `float`). By default, LLDB evaluates expressions using Clang, which parses user-provided code, resolves symbols from DWARF information, and compiles the expression to LLVM IR, which is then just-in-time compiled using LLVM and executed within the debugee's address space.

While powerful, this Clang-based evaluation has key limitations for Hylo. First, complex operations such as function calls may fail due to ABI mismatches between C++ and Hylo. Second, Clang does not enforce Hylo's semantics, potentially leading to unsafe behaviour. Finally, it requires C-like syntax, which might be unfamiliar to some users.

An idea for enabling Hylo expression evaluation in LLDB is to adopt an approach similar to Clang's. Specifically, the Hylo compiler could be reused to compile user-provided expressions. Additionally, it should provide a mechanism to resolve external symbols (e.g., those defined in DWARF DIEs). This approach could be extended further by implementing an **LLDB plugin** [16] for Hylo. In particular, the plugin could handle a custom extension of DWARF, which would be more tailored to represent Hylo-specific constructs, instead of relying on C++ constructs. However, the process of adding a plugin for a new programming language in LLDB is particularly challenging, as seen in the Rust community's effort [26].

5.3 Debugging Dynamic Types in Hylo

Our design supports debugging Hylo's built-in, structure and generic types. If Hylo were limited to these, variable types could be fully resolved at compile time. However, providing **dynamic types** increases expressiveness, even in a language focused on generics. Hylo supports dynamic types through **existentials**, a feature inspired heavily from Swift.

We first introduce **traits**, which define constraints that user-defined structures must satisfy. For example, in Figure 2b, the generic type `T` in the function `add` must conform to the `AdditiveArithmetic` trait, which ensures that addition is supported for the parameters `n` and `m`. Although our current compiler design does not encode traits into the DWARF DIEs, debugging support works properly in code containing traits. That is because traits are resolved before LLVM IR generation, as they are only used for constraint checking.

An **existential type** is a runtime container for any type conforming to specific traits. Such types are denoted with the `any` keyword. For example, `any AdditiveArithmetic` may hold an `Int32` at runtime. Existentials allow constructs like trait-constrained heterogeneous lists, at the cost of runtime indirection due to dynamic dispatch.

In Clang-generated DWARF information, we noticed that each variable's type DIE reflects its static type, even in the case of polymorphic classes. Still, LLDB can determine the **dynamic type** of a variable by using a `LanguageRuntime` plugin [16], which inspects a variable's metadata (e.g., vpointer/vtable if the C++ program conforms to the Itanium ABI [27]). Swift's LLDB fork [18] uses a similar plugin to inspect the **witness table** for existential types, a mechanism that could also be adapted to Hylo. To enable this, traits must also be encoded into DWARF, to be given as type DIEs for existential container variables.

6 Responsible Research

This section outlines the practices we followed to ensure our research was conducted responsibly. In particular, we focus on three areas we believe are most relevant for our work.

6.1 Reproducibility of Our Approach

Our methodology, described in Section 3, is observation-driven, guided by how Clang emits debug information. While the DWARF format is standardized [4], debug information emission remains implementation-dependent. Throughout this work, we used Clang 17.0.0. Although behaviour may vary in future versions, we ensured the emitted debug info for the example in Figure 2a aligns with the DWARF specification.

We manually tested our prototype using a recent version of LLDB (47addd4), and used `llvm-dwarfdump` from LLVM 14 to extract DWARF entries, such as the one from Figure 7. Although these figures illustrate some platform-dependent values of DIE attributes, they do not influence our approach, as they are emitted by LLVM in a platform-independent manner. For completeness, we compiled all programs on an x64 Ubuntu system.

The specific Hylo compiler version we extended is (0951941). Additionally, our implementation, which adds DWARF emission to the Hylo compiler, is made publicly available.[1]

6.2 Transparency of Scope and Limitations

Throughout this report, we explicitly state the Hylo features that our design targets (e.g., see Section 3), and our explanation in Section 4 aims to cover each of them in a comprehensive manner. In practice, our implementation correctly handles additional features beyond this set, but we chose to be conservative in our claim, limiting our scope to those we studied in detail. As such, the reader can assume our scope is limited to the explicitly listed features.

Additionally, we dedicate an entire section to limitations (i.e., Section 5). In particular, we chose to discuss cases that go beyond minor extensions but require substantial engineering effort. Moreover, it is important to emphasise that our implementation is a prototype, intended to demonstrate the feasibility of our approach, not to serve as a production-ready implementation. Debug information is complex, and even mature compilers do not always emit fully correct metadata. Instead, our goal was to lay a solid foundation, which could be extended with additional engineering effort.

6.3 Use of Generative AI Tools

We used generative AI tools in limited, well-defined contexts. Importantly, we **did not** use these tools to generate research ideas, validate technical solutions, write code, or produce full sections of this report. Their use was restricted to two specific cases:

1. **Polishing parts of the report:** We used generative models to refine **existing text**, such as rephrasing small sections for clarity, or checking grammar and spelling. Additionally, we also used them for LaTeX-related tasks, including table formatting and figure alignment.

[1]Our version of the Hylo compiler capable of emitting DWARF information is available at: `https://gitlab.tudelft.nl/jsreinders/pl-tooling-for-hylo/-/tree/debugging-hylo/debuggers?ref_type=heads`

2. **Navigating complex codebases:** When analysing large codebases like LLVM, Clang, or LLDB, we occasionally used generative models to suggest where certain functionality might be implemented (e.g., relevant files or classes). We then manually inspected the source code in these locations and critically assessed whether they matched what we were looking for.

In all cases, outputs from generative models were treated as **unverified suggestions**, meaning that every suggestion we took into account was manually reviewed and validated.

7 Conclusions and Future Work

This work explored the process of adding debugging support for **Hylo**, an emerging systems programming language focused on efficiency, simplicity and safety. We designed and implemented a version of the Hylo compiler capable of emitting DWARF debug information, enabling source-level debugging in LLDB.

Our approach was incremental and observation-driven. By studying how Clang emits DWARF metadata for C++, we adapted similar strategies for Hylo, adjusting for its semantics and compiler architecture. We demonstrated that a significant subset of Hylo features (e.g., variables, functions, user-defined types, and generics) can be debugged effectively without extending the DWARF standard or LLDB.

We also examined the limitations of our approach and outlined directions for future work. In particular, we highlighted the complexity of accurately representing Hylo's projections and fine-grained variable lifetimes in DWARF, as well as the challenges of supporting rich expression evaluation in LLDB and debugging Hylo's dynamic types (i.e., existentials), which require custom LLDB plugins or even a dedicated LLDB fork.

Looking forward, several engineering and research directions can extend this work. Our current design could be refined and upstreamed into the main Hylo compiler, making debugging accessible to all users. Integration with IDEs via protocols such as the Debug Adapter Protocol [28] would improve usability for developers unfamiliar with command-line tools. Additional opportunities include supporting other debuggers (e.g., GDB[29], CDB [30]) and investigating how Hylo's concurrency model interacts with debugging concurrent programs.

Beyond Hylo, this work illustrates a general and practical methodology for incrementally adding debugging support to new LLVM-based programming languages. We believe our approach lays the foundation for making emerging systems languages more approachable, debuggable, and ready for adoption in real-world software development.

References

[1] Abdulaziz Alaboudi and Thomas D LaToza. "An exploratory study of debugging episodes". In: *arXiv preprint arXiv:2105.02162* (2021).

[2] Ryan Fleury. *Demystifying Debuggers, Part 3: Debugger-Kernel Interaction.* 2024. URL: `https://www.rfleury.com/i/153647066/debug-event-apis`.

[3] Sy Brand. *Building a Debugger: Write a Native x64 Debugger From Scratch.* No Starch Press, 2025. ISBN: 978-1-71850-408-0.

[4] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format Version 5.* Tech. rep. Accessed: 2025-05-23. DWARF Standards Committee, Feb. 2017. URL: `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] Alexander A. Stepanov and Daniel E. Rose. *From Mathematics to Generic Programming.* 1st ed. Addison Wesley, 2015. ISBN: 978-0321942043.

[7] Dimitri Racordon et al. “Implementation Strategies for Mutable Value Semantics.” In: *J. Object Technol.* 21.2 (2022), pp. 2–1.

[8] LLVM Project. *The LLDB Debugger.* Accessed: 2025-05-23. 2025. URL: `https://lldb.llvm.org/index.html`.

[9] The LLVM Project. *Clang: a C Language Family Frontend for LLVM.* Accessed: 2025-06-20. n.d. URL: `https://clang.llvm.org/`.

[10] The Linux Foundation. *The Linux Kernel Documentation.* `https://www.kernel.org/doc/html/latest/`.

[11] *fork(2) - Linux manual page.* Accessed: 2025-05-02. n.d. URL: `https://man7.org/linux/man-pages/man2/fork.2.html`.

[12] *exec(3) - Linux manual page.* n.d. URL: `https://man7.org/linux/man-pages/man3/exec.3.html`.

[13] *waitpid(3p) - Linux manual page.* n.d. URL: `https://man7.org/linux/man-pages/man3/waitpid.3p.html`.

[14] *ptrace(2) - Linux manual page.* n.d. URL: `https://man7.org/linux/man-pages/man2/ptrace.2.html`.

[15] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developerâs Manual Volume 2 (2A, 2B, 2C, 2D): Instruction Set Reference, A-Z.* Intel Corporation. 2025.

[16] LLVM Project. *Adding Programming Language Support.* Accessed: 2025-05-23. LLVM Project. 2025. URL: `https://lldb.llvm.org/resources/addinglanguagesupport.html#adding-programming-language-support`.

[17] LLVM Project. *Finish the Language Abstraction and Remove All the Unnecessary APIs.* Accessed: 2025-05-23. LLVM Project. 2025. URL: `https://lldb.llvm.org/resources/projects.html#finish-the-language-abstraction-and-remove-all-the-unnecessary-api-s`.

[18] Apple Inc. *LLDB for Swift.* 2025. URL: `https://github.com/swiftlang/llvm-project/tree/next/lldb`.

[19] The Hylo Group. *The Hylo Programming Language.* Accessed: 2025-05-23. The Hylo Group. 2025. URL: `https://www.hylo-lang.org/`.

[20] LLVM Project. *Source Level Debugging with LLVM.* Accessed: 2025-06-02. 2025. URL: `https://llvm.org/docs/SourceLevelDebugging.html`.

[21] ISO/IEC JTC1/SC22/WG21. *C++ Draft International Standard.* Tech. rep. N4860. Accessed: 2025-06-22. International Organization for Standardization, Feb. 2020. URL: `https://isocpp.org/files/papers/N4860.pdf`.

[22] Michael Matz et al. “System v application binary interface”. In: *AMD64 Architecture Processor Supplement, Draft v0* 99.2013 (2013), p. 57.

[23] LLVM Project. *How To Update Debug Info: A Guide for LLVM Pass Authors.* Accessed: 2025-06-16. 2025. URL: `https://llvm.org/docs/HowToUpdateDebugInfo.html`.

[24] The Hylo Group. *Language Tour.* Accessed: 2025-05-23. The Hylo Group. 2025. URL: `https://docs.hylo-lang.org/language-tour`.

[25] The Hylo Group. *Hylo Language Specification.* Accessed: 2025-05-23. The Hylo Group. 2025. URL: `https://github.com/hylo-lang/specification/blob/main/spec.md`.

[26] walnut356. *LLDB's TypeSystems: An Unfinished Interface.* Accessed: 2025-06-22. 2025. URL: `https://walnut356.github.io/posts/lldbs-typesystems-an-unfinished-interface/`.

[27] Itanium C++ ABI Committee. *Itanium C++ ABI.* Accessed: 2025-05-31. 2017. URL: `https://itanium-cxx-abi.github.io/cxx-abi/abi.html`.

[28] Microsoft. *Debug Adapter Protocol - Specification.* Accessed: 2025-06-22. 2024. URL: `https://microsoft.github.io/debug-adapter-protocol/specification`.

[29] GNU Project. *GDB: The GNU Project Debugger.* Accessed: 2025-05-23. 2023. URL: `https://sourceware.org/gdb/`.

[30] Microsoft. *Debugging a User-Mode Process Using CDB.* Accessed: 2025-06-22. 2021. URL: `https://learn.microsoft.com/en-us/windows-hardware/drivers/debugger/debugging-a-user-mode-process-using-cdb`.