International Journal of Research Publication and Reviews

Journal homepage: www.ijrpr.com ISSN 2582-7421

Flow-Wing: Performant Hybrid-Type Programming Language

Kushagra Rathore

Department of Computer Science and Engineering, Medi-caps University, Indore (M.P.), India

ABSTRACT—

Flow-Wing, a novel programming language developed using LLVM and C++, offers a unique blend of performance and flexibility. Named for its fluidity and agility, Flow-Wing stands out in the realm of programming languages by integrating both static and dynamic typing systems. This hybrid approach caters to the increasing demand for adaptable languages, allowing developers to choose their preferred typing style based on project requirements. Leveraging LLVM optimization, Flow-Wing achieves remarkable speed and efficiency, surpassing languages like Java and Python. Furthermore, Flow-Wing extends its reach to the web, enabling developers to run their code directly in browsers through WebAssembly (Wasm). This paper delves into the design and implementation of Flow-Wing, positioning it as a promising addition to the programming language landscape.

Keywords— Flow-Wing, LLVM, Hybrid Typing

I. Introduction

This paper introduces Flow-Wing, a novel compiler/language developed using LLVM and C++, designed to offer a unique blend of performance and flexibility. Named for its fluidity and agility, Flow-Wing is a standout in the realm of programming languages, offering a hybrid typing system and superior performance, thanks to LLVM optimization.

Flow-Wing's comprehensive compiler pipeline encompasses all stages of compilation, from input parsing and lexical analysis to syntax tree generation and LLVM intermediate representation (IR) generation. Its unique selling point lies in its integration of both static and dynamic typing systems. This feature caters to the increasing demand for hybrid typing languages, providing developers with the flexibility to select their preferred typing system based on their project requirements. The hybrid typing system of Flow-Wing not only boosts developer productivity but also broadens the scope for crafting efficient and adaptable software solutions.

Leveraging LLVM as its backend, Flow-Wing achieves remarkable speed and efficiency, outperforming languages such as Java and Python. By utilizing LLVM's advanced optimization techniques, Flow-Wing optimizes code generation and execution, leading to faster execution times and enhanced overall performance.

One of Flow-Wing's key features is its ability to incorporate both static and dynamic typing systems. This flexibility allows users to write code in static, dynamic, or hybrid typing styles, with the compiler adjusting the generated LLVM IR to match.

Moreover, Flow-Wing extends its functionality to the web, offering integration with web browsers. By generating LLVM IR that can be compiled down to WebAssembly (Wasm), Flow-Wing empowers developers to run their code directly in the browser environment, thereby enhancing its accessibility and versatility. This feature underscores Flow-Wing's commitment to meeting the evolving needs of today's developers.

II. Literature Review

In recent years, the landscape of programming languages has witnessed a significant shift towards hybrid type systems, blending aspects of dynamically and statically typed languages. Additionally, languages leveraging the LLVM (Low Level Virtual Machine) compiler infrastructure have demonstrated superior performance compared to those that do not. This literature survey explores the rationale behind the development and adoption of hybrid type languages, as well as the factors contributing to the enhanced speed of LLVM-based languages, supported by relevant research in the field.

Efficient resource utilisation and performance optimization are critical considerations in modern software development. Statically typed languages, by virtue of their compile-time optimizations and type inference mechanisms, often outperform dynamically typed counterparts in terms of execution speed and memory usage. However, the rigid type system of statically typed languages can sometimes impose unnecessary overhead and hinder productivity, particularly in scenarios requiring rapid iteration and experimentation. Hybrid type languages address this dilemma by offering a pragmatic compromise, allowing developers to selectively enforce type constraints where performance is paramount while retaining the flexibility of dynamic typing in less performance-critical sections of code [8].

The productivity of software developers is intricately linked to the expressiveness and ease of use of the programming languages they employ. Dynamically typed languages excel in rapid prototyping and iterative development scenarios, enabling developers to quickly experiment with ideas and iterate on solutions without being encumbered by the strictures of static typing. However, as projects scale in size and complexity, the absence of compile-time type checks and documentation can impede code maintenance and collaboration efforts. Hybrid type languages mitigate these concerns by offering optional type annotations and gradual typing mechanisms, allowing developers to incrementally introduce static typing into existing codebases and reap the benefits of improved tooling support and code readability without sacrificing the dynamic nature of their workflows [9].

Empirical studies and adoption trends corroborate the utility and efficacy of hybrid type programming languages in addressing the aforementioned challenges. Projects such as TypeScript, a superset of JavaScript with optional static typing, have gained widespread adoption in both industry and open-source communities, owing to their ability to provide a smooth migration path from dynamically typed codebases while offering tangible benefits in terms of code maintainability and developer productivity. Similarly, languages like Kotlin and Swift incorporate hybrid type features to cater to the diverse needs of developers working on large-scale software projects spanning multiple domains and platforms [10].

The LLVM project, initiated by Chris Lattner at the University of Illinois at Urbana-Champaign, has revolutionised compiler technology and software optimization. LLVM provides a modular compiler infrastructure with a suite of reusable components, including a highly optimised intermediate representation (IR), a just-in-time (JIT) compiler, and a wide range of optimization passes. Languages that leverage LLVM as their compilation backend, such as Rust, Swift, and Julia, benefit from LLVM's advanced optimization capabilities, resulting in faster execution speeds and reduced memory footprint. LLVM's aggressive optimization passes, including loop vectorization, inlining, and target-specific optimizations, contribute to the superior performance of LLVM-based languages compared to those that do not utilise LLVM [11].

LLVM's modular architecture enables efficient code generation tailored to specific target architectures, ranging from desktop CPUs to embedded systems and GPUs. LLVM's ability to generate optimised machine code for diverse hardware targets contributes to its versatility and performance across a wide range of platforms. In contrast, languages like Java and Python typically rely on bytecode interpretation or Just-In-Time (JIT) compilation techniques for execution. While JIT compilation can improve performance by dynamically optimising frequently executed code paths, it introduces overheads such as runtime compilation and interpretation, which may limit overall performance and responsiveness, particularly in latency-sensitive applications [12].

III. Methodology

The development of a programming language is a multifaceted endeavor that requires meticulous planning, innovative design, and rigorous implementation. In this research paper, we delve into the methodology employed in creating Flow-Wing, a performant hybrid-type programming language. Flow-Wing's unique features, such as its hybrid typing system and exceptional performance attributed to LLVM optimization, position it as a promising addition to the programming language landscape.

a) Design and Implementation of the Flow-Wing Compiler

The Flow-Wing compiler is written in C++ and utilizes the LLVM backend to generate LLVM Intermediate Representation (IR), which is then compiled down to binary, object, or executable files. The compiler pipeline encompasses all steps from input parsing and lexical analysis to syntax tree generation and LLVM IR generation.

The Flow-Wing compiler takes Flow-Wing code files as input with the *".fg"* extension. The compiler then goes through several phases to convert the input code into an executable program. One of the distinguishing features of Flow-Wing is its hybrid typing system, which allows users to write code in either static, dynamic, or hybrid typing styles. The compiler adapts the generated LLVM IR accordingly based on the chosen typing system.

In addition to its hybrid typing system, Flow-Wing leverages LLVM as its backend to achieve exceptional speed and efficiency. By harnessing LLVM's advanced optimization techniques, Flow-Wing optimizes code generation and execution, resulting in faster execution times and improved overall performance.

b) Lexical Analysis

The lexical analysis phase is the first phase of the compiler pipeline. In this phase, Flow-Wing converts the program text into words and tokens. For example, the statement *var x: int = 2* is broken up into the following tokens:

var: Identified as the VarKeyword, indicating the declaration of a variable.

x: Represents an IdentifierToken, serving as the symbolic name assigned to the variable.

:: Functions as the ColonToken, used to separate the variable name from its declared type.

int: Designated as the Int32Keyword, specifying the data type of the variable as a 32-bit integer.

=: Denoted as the EqualsToken, indicating an assignment operation.

2: Represents a NumberToken, specifying the value assigned to the variable, in this case, the integer 2.

These tokens are represented as instances of the SyntaxToken class. The SyntaxToken class holds the position, text, and value of the token, which can be later used in creating the syntax tree.

```
std::make_unique<SyntaxToken<std::any>(getAbsoluteFilePath(),          this->lineNumber, PlusToken,this->position++,"+");
```

Fig. 1. FlowWing syntax token class

The Lexer::readSymbol() method is used to read symbols and create the corresponding SyntaxToken instances. For instance, the '+' symbol is read and a SyntaxToken instance with SyntaxKind::PlusToken is created.

c) Recursive-Descent Parsing

Recursive-descent parsing, also known as top-down parsing, is one of the simplest and most widely used parsing techniques. The fundamental concept of recursive-descent parsing is to associate each non-terminal with a procedure. The objective of each procedure is to read a sequence of input characters that can be generated by the corresponding non-terminal and return a pointer to the root of the parse tree for the non-terminal. The structure of the procedure is dictated by the productions for the corresponding non-terminal [7].

The procedure attempts to "match" the right-hand side of some production for a non-terminal. To match a terminal symbol, the procedure compares the terminal symbol to the input; if they agree, then the procedure is successful and consumes the terminal symbol in the input (i.e., moves the input cursor over one symbol). To match a non-terminal symbol, the procedure simply calls the corresponding procedure for that non-terminal symbol (which may be a recursive call, hence the name of the technique) [1].

d) Flow-Wing Parsing Algorithm

Flow-Wing's parsing algorithm is a manifestation of the recursive-descent parsing technique. Below is a pseudo-code representation of the core parsing algorithm used in Flow-Wing:

```
function parseExpression(parentPrecedence)
  left = null
  unaryOperatorPrecedence = getCurrent().getUnaryOperatorPrecedence()
  if unaryOperatorPrecedence != 0 and unaryOperatorPrecedence >= parentPrecedence then
    operatorToken = match(getCurrent().getKind())
    operand = parseExpression(unaryOperatorPrecedence)
    left = new UnaryExpressionSyntax(operatorToken, operand)
  else
    left = parsePrimaryExpression()
  while true do
    precedence = getCurrent().getBinaryOperatorPrecedence()
    if precedence == 0 or precedence <= parentPrecedence then
      break
    operatorToken = match(getCurrent().getKind())
    right = parseExpression(precedence)
    left = new BinaryExpressionSyntax(left, operatorToken, right)
  return left
```

Fig, 2. Flow-Wing parsing algorithm example

The parsing process is a critical component of the compiler pipeline, and Flow-Wing employs a recursive-descent parsing technique to generate the parse tree or abstract syntax tree (AST) [2].

Parsing of Unary and Primary Expressions: The parseExpression method, which forms the core of the parsing logic within the FlowWing language, commences by evaluating the precedence of unary operators. In the event that a unary operator is detected and its precedence equals or surpasses the parentPrecedence, a unary expression is parsed. In the absence of a unary operator, or if the unary operator's precedence is lower than the parentPrecedence, a primary expression is parsed.

Construction of Binary Expressions: Following the parsing of unary and primary expressions, the parseExpression method iterates through binary operators. During this process, binary expressions are constructed in accordance with their precedence and associativity. This iterative procedure ensures the accurate arrangement of expressions within the syntax tree.

e) Generation and Analysis of Syntax Tree for Prime Number Determination Algorithm

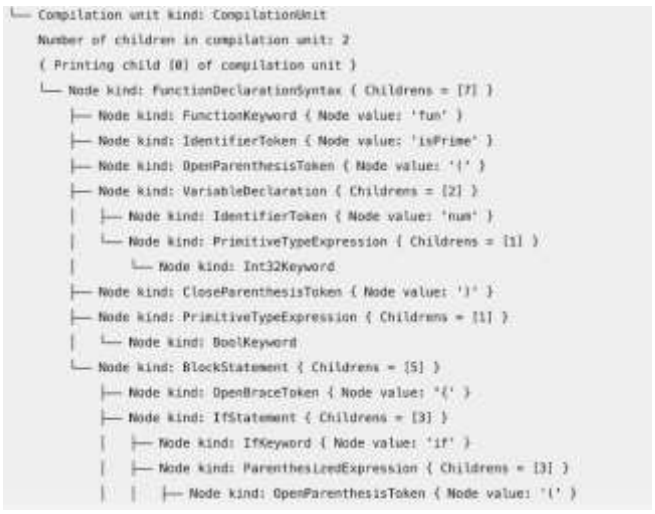
This section presents the methodology employed in the study, focusing on the generation and interpretation of syntax trees using the Flow-Wing system. The methodology is demonstrated using the isPrime function, a simple algorithm that checks if a given number is prime.

Program Description: The isPrime function is a straightforward algorithm that takes an integer as input and returns a boolean value indicating whether the number is prime or not [3]. The function is defined as follows:

```
fun isPrime(num : int ) -> bool {
  if(num <= 1) {
    return false
   }
  for var i = 2 to num : 1 {
   if ( num % i == 0) {
     return false
    }
  }
  return true
}
```

Fig. 3. FLowWing program, function to check number is prime or not

Syntax Tree Generation: Flow-Wing generates a syntax tree for the isPrime function. The syntax tree provides a detailed view of the program's structure, demonstrating the clarity and efficiency of Flow-Wing's syntax. The root of the syntax tree is the Compilation Unit, which contains all the elements of the program.



```
└── Compilation unit kind: CompilationUnit
    Number of children in compilation unit: 2
    ( Printing child [0] of compilation unit )
    └── Node kind: FunctionDeclarationSyntax ( Childrens = [7] )
        ├── Node kind: FunctionKeyword ( Node value: 'fun' )
        ├── Node kind: IdentifierToken ( Node value: 'isPrime' )
        ├── Node kind: OpenParenthesisToken ( Node value: '(' )
        ├── Node kind: VariableDeclaration ( Childrens = [2] )
        │   ├── Node kind: IdentifierToken ( Node value: 'num' )
        │   └── Node kind: PrimitiveTypeExpression ( Childrens = [1] )
        │       └── Node kind: Int32Keyword
        ├── Node kind: CloseParenthesisToken ( Node value: ')' )
        ├── Node kind: PrimitiveTypeExpression ( Childrens = [1] )
        │   └── Node kind: BoolKeyword
        └── Node kind: BlockStatement ( Childrens = [5] )
            ├── Node kind: OpenBraceToken ( Node value: '{' )
            ├── Node kind: IfStatement ( Childrens = [3] )
            │   ├── Node kind: IfKeyword ( Node value: 'if' )
            │   ├── Node kind: ParenthesizedExpression ( Childrens = [3] )
            │   │   ├── Node kind: OpenParenthesisToken ( Node value: '(' )
```

Fig. 4. syntax tree of the isPrime function

f) Code Generation and LLVM

LLVM, short for Low-Level Virtual Machine, represents a pivotal advancement in compiler technology. Originally conceived as a research project at the University of Illinois, LLVM has evolved into a robust and widely adopted compiler infrastructure. Its architecture, characterized by modularity and extensibility, underpins its ability to serve diverse programming languages, platforms, and optimization requirements [4].

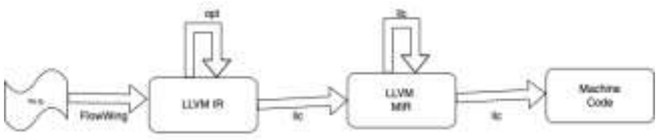


Fig. 5. FlowWing compiler frontend using llvm

LLVM Architecture: At the heart of LLVM lies its intermediate representation (IR), a platform-independent abstraction of low-level code. LLVM's architecture revolves around this IR, enabling efficient code transformations and optimizations. The modularity of LLVM facilitates the construction of a comprehensive compilation pipeline comprising various stages, including frontends, optimizations, and backends [5].

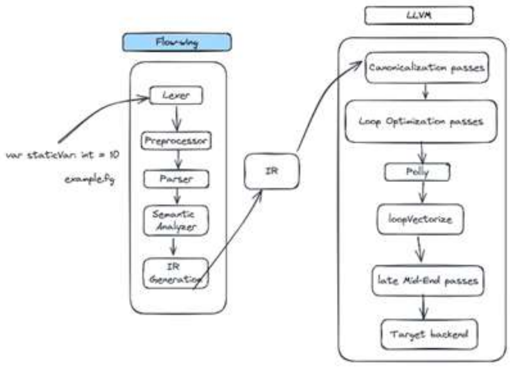


Fig. 6. FlowWing and llvm compiler architecture

Flow-Wing's Use of LLVM Backend for IR Generation: Flow-Wing utilizes LLVM, LLVM-IR, and C++ to develop its compiler/language. It implements a comprehensive compiler pipeline covering input parsing, lexical analysis, syntax tree generation, and LLVM intermediate representation (IR) generation.

Generating LLVM IR for Variable Declaration:

Consider the Flow-Wing code snippet: *var x: int = 10,* This statement can be translated into LLVM IR as follows

```
%x = alloca i32, align 4

store i32 10, i32* %x, align 4
```

Fig. 7. llvm IR

The *%x = alloca i32, align 4* instruction allocates memory for variable x of type *i32* with alignment 4. The store *i32 10, i32* %x, align 4* instruction stores the value "10" into the memory location allocated for x.

g) Hybrid Typing System

One of the key distinguishing features of Flow-Wing is its incorporation of both static and dynamic typing systems. Users have the flexibility to write code in either static, dynamic, or hybrid typing styles, with the compiler adapting the generated LLVM IR accordingly [6].

Dynamic Typing: In Flow-Wing, a variable declared without specifying a type (e.g., var x = 2) is considered dynamic. This means we can assign a string, decimal, boolean, or number to it. For instance:

```
var x = 2

x = "Hello"

x = 3.3

x = false
```

Fig. 8. FlowWing dynamic typed code

When a dynamic variable is created, it is stored as a structure in memory. This structure has two variants:

Dynamic Typing with Structured Memory Allocation: In this variant, which is mostly used in the language, we create four fields for the struct in memory: the first field holds an int32, the second stores a double, the third field stores a bool, and the fourth holds a pointer to a string.

For example, when we write *var x = 3.3*, the Flow-Wing compiler allocates memory to the structure and fills this decimal field value. It also stores a compile-time type variable associated with x, which tracks the current type of x. Whenever we change or assign a different type value to x, the type variable also updates itself.

Here is an example of Flow-Wing code and its corresponding LLVM IR.

```
%DynamicType = type { i32, double, i1, ptr }

define i32 @flowmain() {

entry:
  store i32 2, ptr @x, align 4

  br label %returnBlock

returnBlock: ; preds = %entry

  ret i32 0

}
```

Fig. 9. Flow-Wing code and IR for dynamic variable

Dynamic Typing with Pointer-Based Memory Allocation: The second variant uses a pointer. When we create a variable in Flow-Wing like var x = 2, Flow-Wing represents this variable as a pointer to the location where the actual variable and its type are stored. If we change the type of x by assigning x = “hello”, then Flow-Wing will create another pointer, which will point to the location in memory where the type, which is a string, and a pointer to the string “hello” are stored.

Static Typing: For the static type system, Flow-Wing uses LLVM’s C++ APIs to generate the LLVM IR. For example, for var x:int = 2, its LLVM IR would be:

```
define i32 @flowmain() {

entry:

  store i32 2, ptr @x, align 4

  br label %returnBlock

returnBlock:                              ; preds = %entry

  ret i32 0

}
```

Fig. 10. llvm-ir of flow-wing static type variable

h) Web Integration

In addition, Flow-Wing extends its reach to the web, offering integration with web browsers. By generating LLVM IR that can be compiled down to WebAssembly (Wasm), Flow-Wing enables developers to run their code directly in the browser environment, further expanding its accessibility and versatility.

IV. Results

This section presents the benchmarking results comparing the execution performance of the factorial function implemented in Flow-Wing, a Performant Hybrid-Type Programming Language built using LLVM and C++, against Java, a widely-utilized programming language known for its portability and object-oriented paradigm.

A. Execution Time Analysis:

The benchmarking study compared the execution performance of the factorial function across three programming environments: Flow-Wing, Java, and JavaScript (Node.js). The factorial function, known for its computational intensity, was meticulously evaluated, and the following comprehensive statistics were recorded:

Benchmarking Results of Factorial Function Execution

Performance metrics	**Programming Languages**		
	FLowWing	***Java***	***JavaScript***
Execution Time	0.005 seconds	0.129 seconds	0.074 seconds

CPU Usage	59%	68%	90%
Real Time	0.005 seconds	0.129 seconds	0.074 seconds
User Time	0.00 seconds	0.05 seconds	0.074 seconds
System Time	0.00 seconds	0.04 seconds	0.01 seconds

B. Observations

Flow-Wing Efficiency: Flow-Wing demonstrated exceptional efficiency, completing the factorial computation in a mere 0.005 seconds with minimal CPU usage. This remarkable performance can be attributed to Flow-Wing's integration with LLVM, which optimizes code generation and execution.

Java and JavaScript Comparisons: In contrast, both Java and JavaScript implementations exhibited comparatively longer execution times. Java, renowned for its platform independence, showcased a more prolonged execution time of 0.129 seconds, while JavaScript (Node.js) execution took 0.074 seconds. Despite these differences, both Java and JavaScript implementations still offer robust solutions for various programming needs.

C. Implications

The benchmarking results underscore Flow-Wing's prowess in delivering high-performance solutions, particularly in scenarios necessitating rapid computation and optimal resource utilization. The seamless integration with LLVM empowers Flow-Wing to outperform traditional counterparts like Java and maintain competitive performance against JavaScript implementations.

V. Conclusion

In this research paper, we explored the development of Flow-Wing, a performant hybrid-type programming language. Flow-Wing's incorporation of both static and dynamic typing systems provides developers with unprecedented flexibility. By leveraging LLVM as its backend, Flow-Wing achieves exceptional performance, outperforming established languages. Additionally, its integration with web browsers through WebAssembly enhances accessibility and versatility. As the programming landscape evolves, Flow-Wing stands as a promising choice for developers seeking a balance between performance and adaptability. With its comprehensive compiler pipeline, Flow-Wing is poised to make a significant impact in the world of programming languages.

References

Leermakers, René. The functional treatment of parsing. Vol. 242. Springer Science & Business Media, 2012.

Alfred, V. Aho, S. Lam Monica, and D. Ullman Jeffrey. Compilers Principles, Techniques & Tools. pearson Education, 2007.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to algorithms. MIT press, 2022.

Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

Huber, Joseph, Weile Wei, Giorgis Georgakoudis, Johannes Doerfert, and Oscar Hernandez. "A case study of LLVM-based analysis for optimizing SIMD code generation." In OpenMP: Enabling Massive Node-Level Parallelism: 17th International Workshop on OpenMP, IWOMP 2021, Bristol, UK, September 14–16, 2021, Proceedings 17, pp. 142-155. Springer International Publishing, 2021.

Madsen, Magnus. "Static analysis of dynamic languages." (2015).

Johnstone, Adrian, and Elizabeth Scott. "Generalised recursive descent parsing and follow-determinism." In International Conference on Compiler Construction, pp. 16-30. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998.

Garcia, M., Ortin, F. and Quiroga, J., 2016. Design and implementation of an efficient hybrid dynamic and static typing language. Software: Practice and Experience, 46(2), pp.199-226.

Souza, C. and Figueiredo, E., 2014, April. How do programmers use optional typing? An empirical study. In Proceedings of the 13th international conference on Modularity (pp. 109-120).

Hanenberg, S., Kleinschmager, S., Robbes, R., Tanter, É. and Stefik, A., 2014. An empirical study on the impact of static typing on software maintainability. Empirical Software Engineering, 19(5), pp.1335-1382

Lattner, C. and Adve, V., 2004, March. LLVM: A compilation framework for lifelong program analysis & transformation. In International symposium on code generation and optimization, 2004. CGO 2004. (pp. 75-86). IEEE.

Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R. and Wolczko, M., 1997. Compiling Java just in time. Ieee micro, 17(3), pp.36-43.