Dyna lang whitepaper

Daksh Upadhyay

November 2024

1 Abstract

The Dyna Language (Dyna) is a new programming language inspired by Rust, designed with strict typing an

2 Introduction

TThe Dyna language specification draws direct inspiration from Rust's programming paradigms, focusing on syntax clarity, immutability, and compile-time checking. Although Rust does not possess an official formal specification, its rustc compiler serves as a de facto implementation. Similarly, Dyna's initial specification will guide the development of its MVP (Minimum Viable Product) compiler.

3 Language core Concepts

3.1 Immutability and Statements

Dyna treats variables as immutable by default, reinforcing stability in code behavior and reducing accidental side effects. Variable declarations use the let keyword

```
let variable_name = String::new();
```

Mutable variables require the mut keyword:

```
let mut variable_mutable = String::from("Hello");
```

Statements that are the final expression in a block do not require a semicolon:

```
let variable = if true {
    String::new()
} else {
    String::from("false")
};
```

3.2 Pattern Matching and User-Defined Types

Dyna enables users to define types through enumerations and structs, supporting pattern matching to handle variant data. Enumeration (enum) and match patterns enhance readability and type safety:

```
enum T {
    Some(String),
    None,
}

let var = T::None;

match var {
    T::Some(string) => println!(string),
    T::None => (),
}
```

4 Function Definitions

4.1 Syntax for Function Declarations

Functions in Dyna are defined using the fn keyword. Each parameter's type must be specified using a colon (:), reinforcing type clarity. Dyna's type checker ensures arguments align with declared function types:

```
fn name_of_function(mut arg: String) -> String {
    arg.push_str("value");
    arg
}
```

4.2 Return Type Specification

Functions may specify a return type using the -¿ syntax. If no return type is specified, the function defaults to returning () (unit type).

5 Arrays and Compile-Time Constraints

5.1 Array Declaration

Dyna allows the creation of compile-time arrays, requiring type declarations due to the absence of type inference. An error will be triggered if the type is omitted:

```
let array: &[u8] = [1, 2, 3, 4];
```

6 Type Checking

6.1 Compile-Time Type Verification

Dyna's type checker provides compile-time validation to prevent mismatched types. While advanced type theories (such as covariance and contravariance) are not handled, basic type matching ensures robust type safety. Errors arise when an argument type does not match the function's declared type:

```
fn name_of_function(mut arg: String) -> String {
    arg.push_str("value");
    arg
}

let array: &[u8] = [1, 2, 3, 4];
// Error: Expected `String`, found `&[u8]`
name_of_function(array);
```

7 Conclusion

This specification provides an initial roadmap for implementing the Dyna language's syntax and type-checking features. The focus on immutability, user-defined types, and compile-time type checking will support Dyna's objective of being a predictable, safe language suitable for scalable software development.

8 Future Work

Advanced Type Inference: Exploring type inference algorithms to reduce verbosity. Error Messaging: Developing clear and actionable error messages for developers. Memory Management: Implementing memory safety features akin to Rust's borrowing model.

9 About the Author

Daksh Upadhyay - Creator and designer of the Dyna language.