Inhaltsverzeichnis

1. Introduction

- The goal of this paper is the conception and implementation of a new programming language called Rym
- What exactly is a programming language
  - artificial language
  - turing complete
  - general

1. general pros of programming languages
2. flaws of currently used programming languages
3. specific features of Rym based on 2. and adhearing to 1.

1.1 Problems of current and past programming languages

- Parallelization
- (Uncontrolled) Undefined Behaviour
  - https://blog.sigplan.org/2021/11/18/undefined-behavior-deserves-a-better-reputation
- Side Effect Safety, Non local reasoning, Sand Boxing
  - https://www.reddit.com/r/ProgrammingLanguages/comments/9eqrfy/comment/e5s3vhb/?utm_source=share&utm_medium=web2x&context=3
  - Code that we read is not understandable in isolation. For example, taking C++: `call(foo);`, is `foo` modified? Dunno.
    - https://www.reddit.com/r/ProgrammingLanguages/comments/9eqrfy/comment/e5v3vj5/?utm_source=share&utm_medium=web2x&context=3
  - Solved by: Koka
- Cannot rewind time while debugging
  - Reversible Computation: Extending Horizons of Computing
- Rust solves most of these, but a watered-down version which only loses a bit of efficiency could probably help a lot. Source => Basically what Rym tries to be :)

1.2 Problems of new programming languages

- https://www.quora.com/What-are-the-biggest-problems-with-modern-programming-languages?share=1
- There are too many of them
- No innovation
  - Too conservative, offer no real improvement to what came before
  - No clear gain to switch to this language
- Being specific to one problem
  - do interesting thing X but do not advance in all other places

1.3 Qualities of a good programming language

- (progressivly having to learn new parts / not having to learn everything at once) makes the language easier to learn
  - see talk about type classes, concept is from Swift
  - begin with a simple script like Python and be able to makke it complex over time
- future proof
  - reserved keywords: try, catch, throw, ..
  - allow breaking changes (semantic versioning), Rust good, Python ok, Js/C++ bad, C does not really extend language anymore but rather core libs right?
- Good Package Manager
  - https://www.reddit.com/r/ProgrammingLanguages/comments/zqjf47/a_good_dependency_manager_for_a_new_programming/
  - https://futhark-lang.org/blog/2018-07-20-the-future-futhark-package-manager.html

2. General trends

Before we start to create the concept for Rym we first have to look at what other language have changed in recent years and if we should add these features. This analysis will extend the general TODO defined in TODO: INSERT CHAPTER REF.

2.1 Type Systems

- getting more powerful
- type inference
- Type Classes have been adapted
  - easily extend functionality of uncontrollable 3rd parties
  - Swift protocols
  - C++23 concepts
  - Rust traits
  - Java, C# Interfaces?
- where?
  - JavaScript
    - TypeScript, other one from Facebook
    - Proposal to add type annotations
  - Python
    - added type annotations
  - Rust, Go, Swift
  - auto in C++
  - state of C, C++, C#, Java, php types?

2.2 Null Safety

- solutions based on powerful type systems
  - Rust, Swift, Dart?
  - TypeScript checks for null/undefined
- what are the others up to?

2.3 Functional programming

- C++
  - addition and work on the *functional* module
  - algorithms to work with lists and iterators
- Python

3. Primitive Data Types

Data types that are not defined in terms of other types are called primitive data types. Nearly all programming languages provide a set of primitive data types. Some of the primitive types are merely reflections of the hardware for example, most integer types and others require only a little nonhardware support for their implementation. More complex types can be created by combining primitive types as we will see in Kapitel 4. [1, S. 400]

3.1 Boolean

Boolean types are perhaps the simplest of all types and have been included in most general-purpose languages designed since 1960. They only have two possible values one for true and one for false. Boolean types are often used to represent switches or flags in programs. Although other types, such as integers, can be used for these purposes, the use of boolean types is more readable. C and C++ still allow numeric expressions to be used as if they were boolean. This is not the case in the subsequent languages, Java and C# which is why Rym will not allow it either. [2], [3]

A boolean value could be represented by a single bit, but because a single bit of memory cannot be accessed efficiently on many machines, they are often stored in the smallest efficiently addressable cell of memory, typically a byte. As this detail is trivial Rym does not specify how to store boolean values. [1, S. 404f]

The boolean type in Rym is called *bool* like in Python, PHP, C#, Go, Rust, Swift and many others. It is named after *George Boole* who pioneered the field of mathematical logic. [3]–[7]

3.2 Boolean Operations

A Boolean value may be created using the *true* or *false* literals

```
const var_1 = true
const var_2 = false
```

and is always the result for the comparison binary operators ==, <, <=, >= and >. The comparision operations actually use the literals in their implementation as well, which can be seen in Kapitel 6. There is also the unary not prefix operator represented by ! which allow one to invert a boolens value . The prefix already suggests that this operator must come before the expression it operates on, as the ! can be used as a unary postfix operator to unwrap a value and **TODO: INSERT CHAPTER REF** explains why that is useful.

In Rym the control-flow expressions *if* and *while* use booleans to decide whether some code should be executed or not. How they work and why their syntax tooks like this will be covered in **TODO: INSERT CHAPTER REF**.

```
const condition = true
if condition { /* do something once */ }
while condition { /* do something forever */ }
```

3.3 Numeric Types

3.3.1 Integer

Another very common primitive numeric data type is the integer. The hardware of many computers supports several sizes of integers. These sizes of integers, and often a few others, are supported by some programming languages. As seen in Tabelle 3.1, Java includes four signed integer sizes: byte, short, int, and long. Some languages, for example C, C++ and C# include unsigned integer types, which are types for integer values without signs. Unsigned types are often used for binary data. 8 bit large unsigned integers can for example represent exactly one byte. [1, S. 400]

The types for C and C++ that are represented in Tabelle 3.1 are using the „cstdint” header as the language standards do not specify the sizes for default integer types and leave them up to the implementation. Types from this standard header file are however required to that exact size. [8, S. 0], [9, S. 0]

- Python: size as large as needed
- PHP: int, size platform dependent, 32bit|64bit
- Java
- C, C++:
  - char, short, int, long are not always the same size
  - cstdint header: c++ standard S. 493f.
- Go
- Zig (special case)
  - arbitrary size
  - size 1–65535
  - i{Size} eg. i333
  - u{Size} eg. u8

Tabelle 3.1: Supported integer formats

Size [Bits]	Java	C#	C, C++	Go	Rust
8	byte	sbyte, byte	int8_t, uint8_t	int8, uint8	i8, u8
16	short	short, ushort	int16_t, uint16_t	int16, uint16	i16, u16
32	int	int, uint	int32_t, uint32_t	int32, uint32	i32, u32
64	long	long, ulong	int64_t, uint64_t	int64, uint64	i64, u64
32|64	—	nint, nuint	—	int, uint	—

Size [Bits]	Java	C#	C, C++	Go	Rust
128	—	—	—	—	i128, u128
pointer	—	—	intptr_t, uintptr_t	intptr, uintptr	isize, usize

3.3.2 Float

Generally languages provide floating point data types that adhere to the „IEEE 754 - Floating-Point” arithmetic standard [10] or its ISO adoption „ISO/IEC 60559” [11]. How wide that support is can be seen in Tabelle 3.2.

- IEEE 754
  - https://en.wikipedia.org/wiki/IEEE_754
  - active version is from 2019 [10]
  - https://ieeexplore.ieee.org/document/8766229
  - same as ISO/IEC 60559
- Accessed: 02.01.2023
  - Js: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Number
  - Python: https://docs.python.org/3/library/stdtypes.html#typesnumeric
  - PHP: https://www.php.net/manual/en/language.types.float.php
  - Java: https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html#jls-4.2.3
  - Go: https://go.dev/ref/spec#Numeric_types
  - Rust: https://doc.rust-lang.org/reference/types/numeric.html#floating-point-types
- Accessed: 09.01.2023
  - C#
    - https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types#837-floating-point-types
    - https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/floating-point-numeric-types
  - C, C++
    - C++ „The value representation of floating-point types is implementation-defined.” S. 75
    - standards do not specific float format to use, they just mandate the minimum range and precision
    - https://en.cppreference.com/w/cpp/language/types

Tabelle 3.2: Supported IEEE-754 floating point formats

Size [Bits]	Js/Ts	Python	PHP	Java	C#	C, C++	Go	Rust
32	Number	—	—	float	float	?	float32	f32
64	—	—	—	double	double	?	float64	f64
32|64	—	float	float	—	—	—	—	—

3.3.3 Decimal

- Pythony 09.01.2023
  - decimal.Decimal
  - The decimal module provides support for fast correctly rounded decimal floating point arithmetic.
  - Once constructed, Decimal objects are immutable.
  - https://docs.python.org/3/library/decimal.html

3.4 Character

- Rym:
  - Name: char
  - valid utf-8 character
  - space: 1 byte?

Tabelle 3.3: Character data types

Language	Formats
Js/Ts	not supported
Python	not supported?
PHP	not supported?
Java	char: 16bit unsinged int
C#	?
C++	?
C	?
Go	?
Rust	`char`

- Java: 09.01.2023 https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html#jls-4.2

3.5 String

- Rym:
  - characters array: [char]
  - dynamic characters vector: String

```
const const_string: [char; 12] = "Hello World!"

impl Add for [char] {
  fn add(move self, move rhs: Self) -> Self {
    [..self, ..rhs]
    // or
    mut new_array = ['\0'; self.length + rhs.length]
    new_array[0..self.length] = self
    new_array[self.length..] = rhs
  }
}
```

4. Algebraic Data Types

Definintion

4.1 Product Types

- explain product types
  - https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming
- exist in:
  - Js/Ts (Arrays, Objects, Maps, WeakMaps)
  - Python (Lists, Records, ..)
  - PHP
  - Java
  - C#
  - C++ (Classes, Structs, ..)
  - C (Structs, ..)
  - Go
  - Rust (Structs, ..)

4.2 Sum Types

- explain sum types
  - https://jrsinclair.com/articles/2019/algebraic-data-types-what-i-wish-someone-had-explained-about-functional-programming/
- exist in: (https://en.wikipedia.org/wiki/Algebraic_data_type)
  - C++, Java 15, Rust, TypeScript
  - F#, Haskell, Idris, Kotlin, Nim, Swift

4.2.1 Enums

```
// Declare an enum.
enum Type {
  Ok,
  NotOk,
}

// Declare a specific enum field.
const c = Type.Ok
```

You can override the ordinal value for an `enum`.

`Enum`s can have methods, the same as `structs`. `Enum` methods are not special, they are only namespaced functions that you can call with dot syntax.

An `enum` can be `matched` upon.

```
enum Foo {
  String,
  Number,
  None,
}
test "enum match" {
  const foo = Foo.Number
  const what_is_it = match foo {
    Foo.String ⇒ "this is a string",
    Foo.Number ⇒ "this is a number",
    Foo.None ⇒ "this is a none",
  }
  assert_eq(what_is_it, "this is a number")
}
```

4.2.2 Non-exhaustive enum

A non-exhaustive enum can be created by adding an underscore as the last field. Non-exhaustive means the enum might gain additional variants in the future, so when unpacking the enum it is required to add a fall through case.

```
// TODO: Use #NonExhaustive Attribute insted?
#NonExhaustive
enum Number {
  One,
  Two,
  Three,
  _,
}

test "match on non-exhaustive enum" {
  const number = Number.One
  const result = match number {
    .One ⇒ true,
    .Two | .Three ⇒ false,
    _ ⇒ false,
  }
  assert(result)
  const is_one = match number {
      .One ⇒ true,
      else ⇒ false,
  }
  assert(is_one)
}
```

4.3 Optional Values

The *Option* type represents an optional value: every *Option* is either *Some* and contains a value, or *None*, and does not. *Option* types are very common in Rym code, as they have a number of uses:

- Initial values
- Return values for functions that are not defined over their entire input range (partial functions)
- Return value for otherwise reporting simple errors, where None is returned on error
- Optional struct fields
- Struct fields that can be loaned or „taken”
- Optional function arguments
- Nullable pointers
- Swapping things out of difficult situations

Options are commonly paired with pattern matching to query the presence of a value and take action, always accounting for the *None* case.

- null references, the billion dollar mistake
  - https://www.lucidchart.com/techblog/2015/08/31/the-worst-mistake-of-computer-science/
  - https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/
  - https://en.wikipedia.org/wiki/Tony_Hoare
- Ts
  - `null`, `undefined`
  - can be detected by Ts compiler
- Java, C, C++, ..
- Rust
  - `Option` enum
  - must be unwrapped to use the value
- `Option`s replace null references
  - is a sum type / enum

[12]

5. Functions

Functions are one of the most important building blocks of many programming languages, especially functional ones, TODO.

5.1 Basic functionality

In Rym it will be possible to define a function by using the *func* keyword followed by a name for the function, a comma separated list of parameters and their respective types as well as the return type of the function.

This means that a function with multiple parameters will generally look like this:

```
func name(parameter0: Type0, parameter1: Type1) → ReturnType {
  // ..
}
```

Just like in any *C-style* language this function may be called by specifying the name of the function followed by a list of arguments within parentheses separated by commas. The arguments must match the types of the parameters defined in the function definition. The previous example function would be called like this:

```
const result = name(argument0, argument1)
```

This would assign the return value to the variable result.

5.2 Scope

When a function is called, the variables and values in the current scope are temporarily suspended and a new scope is created for the function. This new scope contains the parameters of the function as variables, initialized with the values of the arguments passed to the function. The function body is then executed within this new scope, and any variables or values defined within the function are only accessible within this scope. When the function execution is complete, the function scope is destroyed and the original scope is restored.

For example, consider the following code:

```
const x = 5

func changeX(x: Int) → Int {
  const y = 10
  return x + y
}

const result = changeX(3) // result = 13
```

When the function changeX is called, a new scope is created containing the parameter newX initialized with the value 3. The variable y is also defined within this scope and initialized with the value 10. The function body is executed, and the value of newX + y (13) is returned. The function scope is then destroyed and the original scope is restored, resulting in the value 13 being assigned to the variable result. The variable y is not accessible outside of the function scope and therefore does not affect the value of x.

5.3 Returning

In the Rym programming language, a function may be defined with a return type, which specifies the type of value that will be returned by the function. The return type is specified after the arrow symbol (->) in the function definition, followed by the function body in curly braces ({}). The function body may contain any number of statements and expressions, and the value of the final expression in the function body will be returned as the result of the function.

For example, consider the following function definition:

```
func multiply(x: Int, y: Int) → Int {
  let result = x * y
  return result
}
```

In this example, the function multiply takes two integer arguments, x and y, and returns the result of their multiplication as an integer. The function body contains a single statement that defines a local variable result and initializes it with the value of x * y. The return statement then specifies that the value of result should be returned as the result of the function.

However, it is not necessary to specify a return statement in every function. If the return type of the function is specified, the value of the final expression in the function body will be returned automatically. For example, the following function is equivalent to the one above:

```
func multiply(x: Int, y: Int) → Int {
  x * y
}
```

In this case, the value of the expression x * y is returned as the result of the function, without the need for a separate return statement. This allows for more concise and readable code, as the return statement is often unnecessary when the function body contains only a single expression.

5.4 Default Parameter Values

6. Polymorphism

Some much older languages are based on the idea that functions, procedures and their respective parameters have unique types. These languages are said to be *monomorphic* (from Greek ‚one shape'), in the sense that every value and variable can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with *polymorphic* (from Greek ‚many shapes') [13, S. 760] languages in which some values may have more than one type. [14, S. 4]

Polymorphic functions are functions whose parameters can have more than one type. Polymorphic types are types whose operations are applicable to values of multiple types.

Polymorphism is supported by

Most often, it is a good idea to give different functions different names, but when different functions conceptually perform the same task on objects of different types, it can be more convenient to give them the same name [13, S. 326]

- *polymorphic*
- compile-time vs run-time polymorphism
  - Rym will only support compile-time as it is simpler to design and implement
  - run-time polymorphism could be added later on

7. Tooling and Ecosystem

- no time to implement them for Rym

7.1 Package Manager

- examples:
  - Python: pip
  - Js/Ts: npm, yarn, deno
  - C: ?
  - C++: ?
  - Rust: cargo

=> rympkg?

7.2 Formatting and Styleguide

7.2.1 Naming Conventions

- checked by compiler => error that stops compilation
  - variable: `variable_name`
  - function: `function_name`
  - struct: `StructName`
  - enum: `EnumName`
  - trait: `TraitName`

7.2.2 Linters

- examples:
  - Python: pep8, ?
  - Js/Ts: not official?, eslint, prettier, ..
  - PHP: ?
  - ..: ?
  - Go: ?, gofmt
  - Rust: builtin, clippy, rustfmt

Precommit scripts can ensure that only correctly formatted code gets comitted.

=> rymfmt, rym fmt

Appendix

Code Examples

```
func main() -> @Io {
  const numbers = [-14, 1, 3, 6, 7, 7, 12]

  const sum = -13
  if const Some([left, right]) = summands(numbers, -13) {
    print(f"Sum of {left} and {right} = {sum}")
  }
  print("Pointers have crossed, no sum found")
}

/// array must be sorted
func summands(array: [i32], sum: i32) -> Option<[usize; 2]> {
  mut low = 0
  mut high = array.len() - 1

  while low < high {
    const current_sum = array[low] + array[high]

    if current_sum == sum {
      return Some([array[low], array[high]])
    } else if current_sum < sum {
      low += 1
    } else {
      high -= 1
    }
  }

  None
}
```

Rym Parser Grammar

```
TODO
```

References

[1] R. W. Sebasta, *Concepts of Programming Languages. 12. Edition.* Essex: Pearson Education, 2019.

[2] „Java Language Specification. Types, Values, and Variables.“ https://docs.oracle.com/javase/specs/jls/se19/html/jls-4.html (zugegriffen 10. Januar 2023).

[3] „C# 7.0 draft specification. Types.“ https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types (zugegriffen 10. Januar 2023).

[4] „PHP Language Reference. Types.“ https://www.php.net/manual/en/language.types.php (zugegriffen 10. Januar 2023).

[5] „The Go Programming Language Specification. Types.“ https://go.dev/ref/spec#Types (zugegriffen 10. Januar 2023).

[6] „The Rust Reference. Types.“ https://doc.rust-lang.org/stable/reference/types.html (zugegriffen 10. Januar 2023).

[7] „Swift Documentation“. https://developer.apple.com/documentation/swift (zugegriffen 10. Januar 2023).

[8] *ISO/IEC 9899:2018. Programming languages – C*. 2018. Verfügbar unter: https://www.open-std.org/JTC1/SC22/WG14/www/docs/n2310.pdf

[9] *ISO/IEC 14882:2020. Programming languages – C++*. 2020. Verfügbar unter: https://www.open-std.org/JTC1/SC22/WG21/docs/papers/2022/n4910.pdf

[10] „IEEE Standard for Floating-Point Arithmetic“, *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 2019.

[11] *IEEE Standards Association*, 2020. https://standards.ieee.org/ieee/60559/10226 (zugegriffen 2. Januar 2023).

[12] P. Wadler und S. Blott, „How to make ad-hoc polymorphism less ad hoc“, in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989.

[13] B. Stroustrup, *The C++ Programming Language. Fourth Edition*. Boston: Addison-Wesley, 2013.

[14] L. Cardelli und P. Wegner, „On understanding types, data abstraction, and polymorphism“, *ACM Computing Surveys (CSUR)*, Bd. 17, 1985.