Facharbeit

Simon Sommer

10.02.2022

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.

See[1] for additional discussion of literate programming.[2]

[1] Vgl. Knuth, Donald (1984): Literate Programming. In: The Computer Journal, Band 27. S. 99-100.

[2] Vgl. Ebd. S. 80.

2. General trends

- jit compilation?

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. Functions

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

3.1 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.

3.2 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.

3.3 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.

3.4 Default Parameter Values

4. 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')[3] languages in which some values may have more than one type.[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 [5]

- *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

[3] Vgl. Stroustrup, Bjarne (2013): The C++ Programming Language. Fourth Edition. Boston: Addison-Wesley. S. 760.

[4] Vgl. Cardelli, Luca; Peter Wegner (1985): On understanding types, data abstraction, and polymorphism. In: ACM Computing Surveys (CSUR), Band 17. S. 471–523.

[5] Vgl. Ebd. Stroustrup. S. 326

5. Primitive Data Types

5.1 Boolean

- work the same in almost all programming languages
- almost always called `true` and `false`
    - Python: `True` and `False`

5.2 Integer

Tabelle 5.1: Supported interger formats

Language	Formats
Js/Ts	not supported
Python	not supported?
PHP	not supported?
Java	?
C#	?
C++	?
C	?
Go	?
Zig	`u1` .. `u128`, `i1` .. `i128`
Rust	`u8`, `u16`, `u32`, `u64`, `u128`; `i8`, `i16`, `i32`, `i64`, `i128`

5.3 Float

All languages provide floating point data types that adhere to the IEEE 754 - Floating-Point arithmetic standard[6] or its ISO adoption *ISO/IEC 60559*[7] by default. As seen in Tabelle 5.2.

- IEEE 754
    - https://en.wikipedia.org/wiki/IEEE_754
    - active version is from 2019[8]
    - https://ieeexplore.ieee.org/document/8766229
    - same as ISO/IEC 60559

Tabelle 5.2: Supported floating point formats

Language	Formats
Js/Ts	`Number`: `binary64`

[6] Vgl. N.N. (2019): IEEE Standard for Floating-Point Arithmetic. In: IEEE Std 754-2019 (Revision of IEEE 754-2008). S. 1–84.

[7] Vgl. N.N. (2020): https://standards.ieee.org/ieee/60559/10226, aufgerufen am 02.01.2023.

[8] Vgl. N.N. (2019): IEEE Standard for Floating-Point Arithmetic. In: IEEE Std 754-2019 (Revision of IEEE 754-2008). S. 1–84.

Language	Formats
Python	
PHP	`float`: platform dependent, usually `binary64`
Java	`Float`: `binary32`, `Double`: `binary64`
C#	`float`: `binary32`, `double`: `binary64`
C++	`float`: `binary32`, `double`: `binary64`, `long double`: `binary128`
C	`float`: `binary32`, `double`: `binary64`, `long double`: `binary128`
Go	`float32`: `binary32`, `float64`: `binary64`
Rust	`f32`: `binary32`, `f64`: `binary64`

- 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
  - C#: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/types#837-floating-point-types
  - C++: https://en.cppreference.com/w/cpp/language/types
  - C: *ISO/IEC 9899:2018*, https://www.open-std.org/JTC1/SC22/WG14/www/docs/n2310.pdf
  - Go: https://go.dev/ref/spec#Numeric_types
  - Rust: https://doc.rust-lang.org/reference/types/numeric.html#floating-point-types

5.4 Character

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

Tabelle 5.3: Character data types

Language	Formats
Js/Ts	not supported
Python	not supported?
PHP	not supported?
Java	?
C#	?
C++	?
C	?
Go	?

Language	Formats
Rust	`char`

5.5 String

```rust
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
  }
}
```

6. Algebraic Data Types

Definintion

6.1 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, ..)

6.2 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

6.2.1 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 Value2 {
  Hundred = 100,
  Thousand = 1_000,
```

```
  Million = 1_000_000,
}
test "set enum ordinal value" {
  // TODO: How to convert enum to int/uint?
  assert_eq(@enumToInt(Value2.Hundred), 100)
  assert_eq(@enumToInt(Value2.Thousand), 1000)
  assert_eq(@enumToInt(Value2.Million), 1000000)
}
```

You can also override only some values.

```
enum Value3 {
  A,
  B = 8,
  C,
  D = 4,
  E,
}
test "enum implicit ordinal values and overridden values" {
  assert_eq(@enumToInt(Value3.A), 0)
  assert_eq(@enumToInt(Value3.B), 8)
  assert_eq(@enumToInt(Value3.C), 9)
  assert_eq(@enumToInt(Value3.D), 4)
  assert_eq(@enumToInt(Value3.E), 5)
}
```

`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.

```
enum Suit {
  Clubs,
  Spades,
  Diamonds,
  Hearts,

  pub fn isClubs(self: Suit) bool {
    self == Suit.Clubs
  }
}
test "enum method" {
  const suit = Suit.Spades
  assert_eq(suit.isClubs(), false)
}
```

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")
}
```

6.2.2 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)
}
```

6.3 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

Wadler, Philip; Stephen Blott[9]

[9] (1989): How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. S. 60–76.

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

Cardelli, Luca; Peter Wegner (1985): On understanding types, data abstraction, and polymorphism. In: ACM Computing Surveys (CSUR), Band 17. S. 471–523.

Knuth, Donald (1984): Literate Programming. In: The Computer Journal, Band 27.

N.N. (2019): IEEE Standard for Floating-Point Arithmetic. In: IEEE Std 754-2019 (Revision of IEEE 754-2008). S. 1–84.

N.N. (2020): https://standards.ieee.org/ieee/60559/10226, aufgerufen am 02.01.2023.

Stroustrup, Bjarne (2013): The C++ Programming Language. Fourth Edition. Boston: Addison-Wesley.

Wadler, Philip; Stephen Blott (1989): How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. S. 60–76.