BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2019
http://excel.fit.vutbr.cz

Yatta - dynamic, functional programming language for GraalVM

Adam Kövári*

Abstract
GraalVM is a relatively new runtime/virtual machine capable of transforming Abstract Syntax Tree (AST) interpreters into highly optimized compilers. GraalVM provides Java API for implementing AST interpreters that dynamically self-rewrite themselves to provide high runtime performance, called Truffle framework [1].
I have designed Yatta language as an experimental language/interpreter built for GraalVM and implemented it using Truffle framework. *Actual implementation of Yatta interpreter is currently in progress and most features demonstrated in this paper are implemented or in the state of proof-of-concept implementation and those which are not yet implemented, are clearly marked so. Additionally, there is a clear path towards first release sketched in the Conclusions section.*
Yatta explores viability of an advanced functional programming language in the GraalVM environment. It delivers advanced features, such as advanced pattern matching, powerful built-in types and data structures, and built-in concurrency. Asynchronous computations are transparent to the programmer and are implemented by the runtime system.
While Yatta is currently an area of active research and development, one of the main goal is to retain qualities necessary for real world usage. One of the core principles of this language must be easy readability and powerful standard library, so that the language can succeed against its competitors both in the GraalVM world and among other functional programming languages.

Keywords: yatta, dynamic, functional, programming language, graalvm, truffle framework

Supplementary Material: *N/A*

*ikovari@fit.vutbr.cz, 0000-0001-8552-6930, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Yatta is a minimalistic, opiniated, (strongly) dynamically typed, strict, functional programming language, with ML-like syntax, for GraalVM polyglot virtual machine (VM).

Its main purpose is to explore the potential of a language that would combine some of the most useful features of functional programming, such as immutable data structures, powerful pattern matching, simple built-in concurrency in a coherent, easy-to-read language for Java Virtual Machine (JVM).

Yatta does not insist on purity and values readability and ease of use above theoretical guarantees of pure functional languages.

Strictly speaking, Yatta does not implement any existing formal model precisely, and it forces its concurrency model built into language itself. This is a trade-off that on one hand eliminates possibility of full user control of the underlying threading implementation, on the other hand it allows for writing boilerplate/dependency free concurrent code with clear syntax and semantics. This approach also eliminates chaotic situations in mainstream programming languages, where concurrency was added later and/or in form of mutually incompatible approaches and libraries.

Yatta aims to solve some of the shortcomings, both practical and theoretical, of existing functional JVM languages, while exploring benefits of implementation of a functional programming language on GraalVM via Truffle framework.

- **Clojure** - doesn't provide an implicit non-blocking IO, nor built-in pattern matching
- **Scala** - very complicated type system due to object oriented paradigm (OOP) combined with functional programming, notoriously slow compilation times, lack of support for built-in language level asynchronous IO / concurrency
- **Eta / Haskell** - pure, lazy evaluation and related memory leaks [2], monads for side-effects
- **Erlang** - although not a JVM language, it is a language with a built in concurrency model in form of actor system. From Yatta's point of view, actor model is still a low-level approach that requires significant control of the user to model concurrent execution

1.1 Motivation

Yatta language has two main reasons to exist. First is to provide a real-world functional programming language that is very easy to use and is not a Lisp language. Lisp languages are not only notoriously famous for the parentheses-overload syntax, but mainly for their abilities in the meta-programming field. Metaprogramming, while useful at times, can be an enemy to ability to understand other people's code. Yatta aims to be very easily readable, written, no matter by whom.

Secondly, it is to abstract users from dealing with non-blocking asynchronous computations and parallelism. While these features are commonly available in other languages nowadays, they are almost exclusively non-native solutions that come in forms of libraries or frameworks and are difficult to integrate with existing codebases. On top of that, dealing with these additional libraries requires conscious effort of the programmer to choose/learn/integrate these libraries into their mindset when writing new code.

GraalVM conveniently provided a relatively easy way to implement a new, high level, programming language without the hassle of low-level native code/bytecode generation. Truffle framework allows writing Yatta interpreter in the Java language as easily as writing an AST interpreter. Furthermore, it provides tools for optimizing the interpreter performance based on runtime profile.

Yatta language has a well-defined list of priorities:

- good readability - simple syntax, few keywords, virtually no boilerplate
- single expression principle - program is always one expression - this enables simpler evaluation and syntax, allows writing simple scripts as well as complex applications
- few types of expressions - `module`, function[1], `case`, `if`, `let`, `do` and `try`/`catch` + `raise`
- simple module system - ability to expose functions for use in other modules, and ability to import them from other modules. Modules are first level values and can be created dynamically
- powerful and efficient built-in data structures with full support for pattern matching
- built-in runtime level non-blocking asynchronous IO
- simple runtime level concurrency, no additional types or data structures necessary
- polyglot language - interoperability with other languages via GraalVM

The motivation and priorities for Yatta language design yield a language that is different than existing functional languages, in both features and level of abstraction it provides, specifically abstraction related to asynchronous and parallel computations.

2. Expressions

Program in Yatta consists always of evaluation of a single expression. In fact, any Yatta program consist of exactly one expression. Note that syntax in few cases is not final and a subject of an active development.

2.1 Basics

Values in Yatta are represented using following syntax:

- string - in quotes: `"hello world"`
- tuple - in parenthesis: `(1, 2, 3)`
- sequence - in brackets: `[1, 2, 3]`
- symbols - preceded by a colon: `:ok`
- dictionary - in curly braces: `{:one = 1, :two = 2}`
- anonymous function(lambda): `\first second -> first + second`
- function application: function name and arguments separated by spaces: `function arg_one arg_two`
- none: `()`

2.2 Definition of aliases in the executed expression

`let` expression allows defining aliases in the executed expressions. This expressions allows evaluating patterns as well, so it is possible to deconstruct a value from a sequence, tuple, dictionary directly, for example:

[1] function does not need a keyword, it is defined by a name and arguments (patterns)

Listing 1. let expression

```
let
    (1, second) = (1, 2)
    pattern     = expression
in
    expression
```

2.3 Sequence of side effects

`do`[2] expressions is used for definition of a sequence of side effecting expressions.

Listing 2. do expression

```
do
    start_time   = Time\now
    (:ok, line) = File\read_line f
    end_time     = Time\now
    printf line
    printf (end_time - start_time)
end
```

2.4 Pattern matching expression

`case` expression is used for pattern matching on an expression.

Listing 3. case expression

```
case File\read_line f of
    (:ok, line)       -> line
    (:ok, :eof)       -> :eof
    err@(:error, _)   -> err
    tuple    # guard expressions below
        | tuple_size tuple == 3 -> :ok
        | true                  -> :ok
    _                 -> (:error, :unknown)
end
```

2.5 Conditional expression

`if` is a conditional expression that takes form:

Listing 4. if expression

```
if expression
then
    expression
else
    expression
```

Both `then` and `else` parts must be defined.

2.6 Module

`module` is an expression representing a set of functions. Modules must have capital name, while packages are expected to start with a lowercase letter.

Listing 5. module expression

```
module package\DemoMmodule
    exports function1, function2
of
function1 = :something
function2 = :something_else
```

[2]do-expression is still in development and the syntax is a subject of change

2.7 Import expression

Normally, it is not necessary to import modules, like in many other languages. Functions from another modules can be called without explicitly declaring them as imported. However, Yatta has a special `import` expression that allows importing functions from modules and in that way create aliases for otherwise fully qualified names.

Listing 6. import expression

```
import
    funone as one_fun
        from package/SimpleModule
in onefun :something
```

2.8 Exception raising & catching expression

`raise`[3] is an expression for raising exceptions:

Listing 7. raise expression

```
raise :bad_arg

# alternatively with a message

raise :bad_arg "Message␣string"
```

`try/catch` is an expression for catching exceptions:

Listing 8. try/catch expression

```
try
    expression
catch
    (:bad_arg, error_msg)   -> :error
    (:io_error, error_msg) -> :error
end
```

3. Pattern Matching and built-in Data Structures

Yatta has a rich set of built-in types, in addition to ability to define custom data types, known as records. Standard types include:

- integer - signed 64 bit number
- float - signed 64 bit floating point number
- big integer - arbitrary-precision integers
- big decimal - arbitrary-precision signed decimal numbers
- byte
- symbol

[3]raise is still in development and the syntax is a subject of change

- char - UTF-8 code point
- string - UTF-8 strings
- tuple
- sequence - constant time access to both front and rear of the sequence
- dictionary - key-value mapping
- none - no value

Records are implemented using tuples and are a local to modules. Their syntax is not defined yet, however, they conceptually allow accessing tuple elements by name, rather than index.

Pattern matching is the most important feature for control flow in Yatta. It allows simple, short way of specifying patterns for the built in types, specifically:

- Simple types - numbers, booleans, symbols
- Tuples & records
- Sequence & reverse sequence, multiple head & tails & their combinations in patterns
- Dictionaries
- `let` expression patterns
- `case` patterns
- Function & lambda patterns
- Guard expressions
- Non-linear patterns [3] - ability to use same variable in pattern multiple times
- Strings and regular expressions
- Underscore pattern - matches any value

Pattern matching, in combination with recursion are the basis of the control flow in Yatta. Yatta supports tail-call optimization, to avoid stack overflow for tail recursive functions.

4. Asynchronous non-blocking IO & Concurrency

Yatta provides fully transparent runtime system that integrates asynchronous non-blocking IO features with concurrent execution of the code. This means, there is no special syntax or special data types representing asynchronous computations. Everything related to non-blocking IO is hidden within the runtime and exposed via the standard library[4], and all expressions consisting of asynchronous expressions[5] are evaluated in asynchronous, non-blocking matter.

Alternative for building asynchronous operations directly into language itself would be development of such library for existing programming language. Unfortunately, such approach has several shortcomings, mainly that such library would have to be adopted by other libraries/frameworks in order to be usable and it would still impose additional boilerplate simply because libraries cannot typically change language syntax/semantics. This is why Yatta provides these features from day one, built into language syntax and semantics and therefore it is always available to any program without any external dependencies. At the same time, putting these features directly on the language/runtime level allows for additional optimizations that could otherwise be tricky or impossible.

The example below shows a simple program that reads line from two different files and writes a combined line to the third line. The execution order is as follows:

1. Read line from file 1, at the same time, read line from file 2
2. After both lines have been read, write file to file 3 and return it as a result of the `let` expression

The important point of this rather simple example is to demonstrate how easy it is to write asynchronous concurrent code in Yatta.

Listing 9. non-blocking IO example

```
let
    (:ok, line1) = File\read_line f1
    (:ok, line2) = File\read_line f2
in
    File\write_line f3 (line1 ++ line2)
```

This allows programmers to focus on expressing concurrent programs much more easily and not having to deal with the details of the actual execution order. Additionally, when code must be executed sequentially, without explicit dependencies, a special expression `do` is available.

In terms of implementation, the runtime system of Yatta can be viewed in terms of promise pipelineing or call-streams [4]. The difference is that this pipelining and promise abstraction as such is completely transparent to the programmer and exists solely on the runtime level.

In terms of parallelization of non IO related code, Yatta will provide several standard library features, which will turn normal functions into runtime-level promises.

5. Evaluation

Evaluation of an Yatta program consists of evaluating a single expression. This is important, because

[4] Asynchronous IO operations in the standard library will be implemented using Java NIO

[5] Asynchronous expression is usually obtained from the standard library or created by function timeout

everything, including module definitions are simple expressions in Yatta.

Module loader then takes advantage of this principle, knowing that an imported module will be a file defining a module expression. It can simply evaluate it and retrieve the module itself.

6. Syntax

Syntax is intentionally very minimalistic and inspired in languages such as SML or Haskell. There is only a handful of keywords, however, it is not as flexible in naming as Haskell for example.

Yatta programs have ambition to be easily readable and custom operators with names consisting of symbols alone are not that useful when reading programs for the first time. Therefore Yatta does not support custom operators named by symbols only.

7. Error handling

Yatta is not a pure language, therefore it allows raising exceptions. Exceptions in Yatta are represented as a tuple of a symbol and a message. Message can be empty, if not provided as an argument to the keyword/function `raise`.

Yatta, as it is running on GraalVM platform needs to support catching underlying JVM exceptions. These exceptions can be caught by fully qualified name of the Java exception class.

Furthermore, Yatta will provide standard functions to extract message from the JVM exceptions, as well as a stacktrace from any exceptions.

Catching exceptions is exactly the same for underlying asynchronous code, with no additional syntax or semantics required. Yatta runtime makes sure exceptions are caught regardless of whether the function being executed is an IO/CPU runtime promise or a basic function.

This makes it easy to write asynchronous and non-blocking code with proper error handling, because to the programmer code always appears exactly the same, as if it were blocking, synchronous code in mainstream languages.

Previous example extended by error handling:

Listing 10. non-blocking IO example - Error handling

```
try
    let
        (:ok, line1) = File\read_line f1
        (:ok, line2) = File\read_line f2
    in
        File\write_line f3 (line1 ++ line2)
catch
    (:match_error, _)  -> :error
    (:io_error, _)     -> :error
end
```

This example is just for demonstration of handling errors when using asynchronous IO code, standard library, including the `file` module is not defined yet.

8. GraalVM & Truffle Framework

GraalVM [5] is a high-performance Virtual Machine with approach that relies on AST interpretation where a node can rewrite itself to a more specialized or more general node, together with an optimizing compiler that exploits the structure of the interpreter [1]. The compiler uses speculative assumptions and deoptimization in order to produce efficient machine code.

Truffle framework is a Java library that allows writing an AST interpreter for a language. An AST interpreter is probably the simplest way to implement a language, because it works directly on the output of the parser and does not involve any bytecode or conventional compiler techniques, but it is often slow. GraalVM has combined it with a technique called partial evaluation [6], which allows Truffle to use GraalVM to automatically provide a just-in-time compiler for the language, just based on the AST interpreter.

Yatta is implemented using Truffle framework and in this way benefits from various just-in-time optimizations available in GraalVM. Yatta interpreter can be used within GraalVM, or distributed as a standalone, ahead-of-time, compiled native binary using SubstrateVM.

In terms of polyglot use, Yatta will not allow use of OOP code directly, mainly because that would require support for syntax for calling methods, creating objects and supporting OOP syntax is something that would compromise the simplicity of the syntax with an unnecessary clutter. Instead, Yatta focuses on defining clear interface for writing wrappers that provide compatible, functional, interface to external code and libraries.

What this means in practice, is that calling a Java library will not be possible directly, without writing a wrapper first. Yatta values simple syntax more than ability to call Java code directly.

9. Conclusions

Yatta is among first functional programming languages being implemented on GraalVM. Perhaps the first one with advanced pattern matching of which the implementation has already presented several interesting challenges, which will be explored in further papers.

Concurrency model in Yatta is interesting in the sense of level of abstraction it provides. Asynchronous and non-blocking nature is built into standard library and requires no interaction or even awareness of the programmer. Parallelization of CPU-bound computations is very simple and presumes knowledge of only a handful of standard library functions. It is built with functions alone, no abstraction, such as processes, message sending or synchronization is required.

I believe Yatta combines interesting, real-world inspired, concepts and presents itself as a bold step forward in the area of dynamic programming languages. Simplicity, ease of use, lack of complicated abstractions, such as monads, can prove to be very useful for writing scripts, applications and complex systems.

This paper presents new dynamic, functional programming language Yatta, for GraalVM. Yatta combines useful properties of functional programming languages, such as immutability or pattern matching, with simple asynchronous computations and built-in runtime-level concurrency model.

Yatta is an experimental language, that hopes to prove that further innovation in dynamic languages is useful for both real-world applications, as well as a research topic involving areas such as concurrent evaluation for example.

Yatta is in active development. Module system, pattern matching is in very advanced status, non-blocking IO and concurrency are currently in the status of a proof of concept. Further development will aim to deliver results in terms of performance in real-world use-cases so that it becomes clear which places need to be optimized more.

Functional programming is often thought in terms of static languages with strong static compile time guarantees. Yatta goes in a different direction and explores potential as a dynamic language with a powerful runtime system. This makes language much simpler to read and write in.

Yatta is currently in active development and there is a clear path towards the first release. I'm working on delivering first usable release along with a standard library later this year.

This includes:

- implementation of records
- implementation of non-blocking IO and concurrency
- exception handling
- enable polyglot use of other languages
- adding additional types, such as big integer / decimal
- additional tests, optimizations and syntax cleanup
- better strings, including string interpolation, regular expressions and pattern matching

These are the features I aim towards completing prior to first public release.

Acknowledgements

I would like to thank my supervisor Prof. Alexander Meduna for his continuous support and guidance, most notably in the area of formal models and theoretical computer science.

I would also like to thank Fedor Gavrilov, who has been essential in the implementation of data structures as well as numerous brainstorming sessions and discussions about variety of concepts of the language.

References

[1] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM.

[2] Neil Mitchell. Leaking space.

[3] R. Ramesh and I. V. Ramakrishnan. Nonlinear pattern matching in trees. *J. ACM*, 39(2):295–316, April 1992.

[4] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *SIGPLAN Not.*, 23(7):260–267, June 1988.

[5] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM.

[6] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. In *ACM SIGPLAN Notices*, volume 52, pages 662–676. ACM, 2017.