Clojerl: The Expressive Power of Clojure on the BEAM

Juan Facorro
Stockholm, Sweden
juan@facorro.com

Natalia Chechina
Bournemouth University
Bournemouth, United Kindgdom
nchechina@bournemouth.ac.uk

Abstract

The development of new features and approaches in programming languages is a continuous and never-ending task, as languages are ultimately tools for expressing and solving problems. The past decade has seen a surge in languages implemented for the BEAM as part of a search to combine the fault-tolerance and scalability of the BEAM with a set of desired language features.

In this paper we present Clojerl, an implementation of the Clojure language with a rich set of data processing capabilities and the expressive power of Lisp for the BEAM. The main design principles of Clojerl are to provide (1) seamless interoperability with the BEAM to enable frictionless interaction with other BEAM languages and (2) portability with Clojure to enable existing Clojure code to run on the BEAM with little or no modifications. We evaluate Clojerl by running a set of experiments that analyse the performance of eight most widely used expressions. While the results of complex expressions show that Clojerl requires further optimisations, Clojerl significantly outperforms Clojure in a set of basic expressions, confirming that Clojerl has the potential to provide a competitive performance while offering a rich set of programming language features.

CCS Concepts: • **Software and its engineering → Functional languages**.

Keywords: BEAM, Clojure, functional programming, programming language, fault tolerance, scalability, concurrency

ACM Reference Format:
Juan Facorro and Natalia Chechina. 2020. Clojerl: The Expressive Power of Clojure on the BEAM. In *Proceedings of the 19th ACM SIGPLAN International Workshop on Erlang (Erlang '20), August 23, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3406085.3409012

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Erlang '20, August 23, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8049-2/20/08…$15.00
https://doi.org/10.1145/3406085.3409012

1 Introduction

In the past couple of decades platforms built around virtual machines and language engines, such as the Java Virtual Machine (JVM), the Erlang VM (BEAM), .NET's Common Language Runtime (CLR), and Javascript's V8 Engine, have grown and matured into ecosystems of tools and libraries. These platforms have also enabled and encouraged the implementation of programming languages other than the official group supported by each platform. As a result, people looking to take the best from the two worlds, combining their language of choice with the power of a specific platform, have created new language implementations.

A programming language is ultimately just a tool for solving problems. When adding a new language to a platform, the language benefits from what the platform's ecosystem has to offer, and the platform benefits from having a new tool available for people to use. A clear example of this is Elixir [11], a programming language implemented on the BEAM, with Ruby-inspired syntax, Erlang-inspired concurrency model, and a number of features inspired by other languages.

This paper presents the addition of another language to the BEAM's tool-belt: Clojerl, an implementation of Clojure on the BEAM. Clojerl is a bridge between the two worlds (i.e. BEAM and Clojure), in an attempt to include the best of both. The resilient and highly available way to build systems in the BEAM, together with the expressive power and rich data processing capabilities of Clojure.

The contributions of this paper are as follows:

- We introduce a new language to the BEAM family called Clojerl (Section 3), an implementation of Clojure, and present the benefits of having a modern Lisp implementation on the BEAM (Sections 4, 5, 6, 7 and 8).
- We evaluate the performance of this implementation by comparing it against the canonical Clojure implementation on the JVM (Section 9).
- We demonstrate that in basic expressions Clojerl outperforms Clojure (experiments #1–#4 in Table 5).
- We identify directions for further improvements and optimizations in Clojerl (Sections 9, 10).

2 Related Work

2.1 Clojure

Clojure is a modern Lisp created in 2007 to provide better tools to deal with concurrency in an established platform, the JVM. It does this by "directly supporting concurrent software development using functional programming techniques, and [...] provides an environment conducive to agility" [4].

Clojure was designed to be a hosted language in order to take advantage of the host platform's existing ecosystem, such as libraries and tools. Therefore the language aims to enable frictionless interaction with the underlying platform, so that programs written in Clojure can seamlessly call and use code written in other languages for the same platform (e.g. Java).

Clojure is implemented as a Lisp to reap all the benefits this family of languages brings, such as:

- almost no syntax: code is built from lists and can be easily manipulated;
- lambda calculus: yields an extremely small core for the language;
- expressive power: through macros it is possible to create custom language constructs that are closer to the problem space;
- homoiconicity: code is data and data is code [14].

Clojure stands out from other existing Lisp languages due to its focus on concurrency, its usage of immutable persistent data structures (Section 5), and its design principle of being a hosted language. Additionally, protocols (Section 6) and multi-methods (Section 7), though not unique to Clojure, enable extensible and flexible polymorphism.

Clojure on the JVM is not the only existing official implementation. There are currently two other officially supported host platforms: ClojureScript [6] for JavaScript engines and ClojureCLR [8] for the CLR.

The goal of ClojureScript is to bring the power of Clojure to the ubiquitous platform that is the browser, as well as to backend JavaScript engines like NodeJS [12]. ClojureScript diverges in some aspects from Clojure on the JVM. For example, it only supports the numerical data types present in JavaScript, which are a subset of what Clojure on the JVM supports. Another important difference is that since most JavaScript environments do not provide support for concurrent programming, it is not possible for ClojureScript to include Clojure's concurrency constructs.

ClojureCLR is an implementation of Clojure for the Common Language Runtime. The CLR is very similar to the JVM in a number of aspects [9], which makes the translation of code and concepts from one platform to another a very simple and straightforward process. The result is an implementation that provides the same capabilities, tools, and features as Clojure on the JVM, and expands the number of available alternatives when choosing a platform for Clojure.

There is no formal definition of the Clojure language, which makes the canonical implementation on the JVM the closest thing to a definition. Some existing official implementations make some trade-offs based on the capabilities of the platform they are using, but are still considered implementations of Clojure. Therefore we have taken this same approach when implementing Clojure on the BEAM.

2.2 BEAM

The BEAM (also known as the Erlang VM) is a platform designed to build highly concurrent, massively scalable and highly-available systems [10]. Its concurrency model is based on independent light-weight processes that share nothing and communicate through message-passing. This concurrency model is very powerful as it allows systems to handle errors and crashes in a safe way by isolating and monitoring failures and recovering from them as necessary.

The native data structures available in the BEAM are all immutable and some even persistent, such as lists and maps. This aligns with the "share nothing" approach to concurrency, since immutable values by definition cannot be modified by other processes or threads. This is why Clojure's focus on concurrency, immutable persistent data structures and the large active community around it, make Clojure a well suited language for running on the BEAM.

The main language used on the BEAM is Erlang [2]. However, in the past 10 years there has been a surge in the implementation of alternative languages, such as Elixir, Lisp Flavoured Erlang, Alpaca, to name just a few. This has proven beneficial for the usage of the BEAM and also for the communities that have been growing around these new languages.

Apart from Clojerl, there are at least two other languages on the BEAM that are part of the Lisp family: Lisp Flavoured Erlang (LFE) [15] and Joxa [7]. Both languages proved to be very useful when learning concepts about language implementation on the BEAM, and both were created with very specific goals in mind, which differ from Clojerl's.

Lisp Flavoured Erlang was born to provide a Lisp syntax for Erlang. It was designed to stay as close to the Erlang semantics as possible. This allows a direct translation from Erlang code into LFE code and also ensures that performance is at least as good as the original.

Joxa was mainly designed as a platform for creating DSLs that could take advantage of the Erlang VM. Joxa's syntax was inspired by Clojure but its creator was not interested in implementing all of Clojure's features.

Even though Elixir is not from the Lisp language family, it took a number of ideas from Clojure. Protocols, macros, and usage of abstractions as a central tool to build the language's data manipulation libraries, are some of the concepts that Elixir borrowed from Clojure. Elixir's source code is another good reference for techniques regarding language implementation on the BEAM. The implementation of macros in Clojerl is heavily inspired by Elixir's implementation.

Finally, it is important to mention Core Erlang, an intermediate representation of Erlang, intended to lie at a level between source code and the intermediate code typically found in compilers [3]. This is the representation we chose to use in the Clojerl compiler, instead of Erlang's abstract format [1]. Core Erlang provides a lower-level representation giving more control to the user when deciding how to compile an expression.

Another advantage of using Core Erlang is that its semantics and language features are more aligned with Clojure's than Erlang's. As an example, single assignment to variables is not a feature that is present in Core Erlang which becomes very useful when using it to implement Clojerl.

3 Clojerl Overview

Clojerl is an implementation of the Clojure [5] language for the BEAM with *two main design principles*.

The first design principle is to *keep Clojerl as close to Clojure as possible in terms of philosophy, features and semantics* to allow code portability between different platforms. This means that a Clojure library or an application built for the JVM, can be adapted to run using Clojerl on the BEAM (and vice-versa) after addressing the platform-dependent bits in the code.

The second principle is *interoperability with the BEAM's native constructs*, to enable seamless interaction with existing libraries and tools in the BEAM ecosystem. A lot of production ready libraries written in other BEAM languages (e.g. Erlang) are already available, and being able to reuse these libraries delivers a lot of value to anyone starting a new Clojerl project. This reusability is possible because all BEAM languages ultimately compile their code down to BEAM modules and functions, and Clojerl provides the mechanism necessary to be able to call any function that can be loaded by the BEAM.

Clojure is composed of a small set of *special forms* (Table 1). The rest of the language is built up using these special forms to define the functions and macros (Section 8) that are part of Clojure's standard library.

Clojerl supports almost all the special forms in Table 1, and these have a one-to-one mapping to the primitives available in Core Erlang [3]. For example, both languages define `let` expressions which bind one or more values, to one or more names. Core Erlang also provides some additional features that are not available in the Erlang language (e.g. `letrec`), but are required to implement some Clojerl language features (e.g. `letfn`). The `letrec` primitive allows to define a set of named functions that can reference each other and themselves. This is the underlying construct that enables creating recursive anonymous functions in Erlang with an expression such as `fun F(X) -> F(X + 1) end`.

Special Forms	Description
`def`	Creates and interns or locates a global var
`if`	Classic conditional
`do`	Evaluates expressions in order and returns the value of the last one
`let`	Evaluates expressions in a lexical context in which the symbols in the binding forms are bound to their respective initialization expressions
`quote`	Yields the unevaluated provided form
`var`	The symbol must resolve to a var, and the Var object itself (not its value) is returned
`fn`	Defines a function
`loop`	loop is exactly like let, except that it establishes a recursion point at the top of the loop
`recur`	Tail recursive call to the current recursion context
`throw`	An expression is evaluated and thrown
`try`	try..catch..finally
`monitor-enter`	JVM synchronization primitive
`monitor-exit`	JVM synchronization primitive

Table 1. Clojure Special Forms

Clojerl does not support only two of the special forms from Table 1: `monitor-enter` and `monitor-exit`, which are related to JVM synchronization primitives for concurrency handling using locks. The BEAM's message-passing model allows for concurrency without the complexity of using explicit locks, and Clojure's philosophy is aligned with the goal of avoiding the usage of explicit locking when dealing with concurrency. Since there is no support in the BEAM for explicitly holding locks on values or processes, there is no place in Clojerl for these two special forms. The reason Clojure still provides these low level lock constructs in its implementation on the JVM, is interoperability with the underlying platform, as it is one of Clojure's guiding principles.

Following this same principle Clojerl introduces the platform-specific special forms shown in Table 2. They are tailored for the BEAM to enable such things as message-passing, bitstring expressions and interacting with Erlang behaviours.

In the following sections we present some important aspects of Clojerl and discuss how they compare to Clojure. In particular we discuss code organisation, mapping of BEAM modules, and mechanisms that allow to easily use Clojerl code from any other BEAM language (Section 4). Available data types and data structures are central to any language, therefore we discuss supported data types and data structures in Clojerl and also provide rationale behind the missing ones (Section 5). We then present Clojerl's main tools for

Special Forms	Description
`receive*`	Receive messages sent to the process
`erl-fun*`	BEAM fun (i.e. `fun m:f/a`)
`erl-binary*`	BEAM binary expression
`erl-list*`	Literal BEAM list
`erl-alias*`	Alias an expression
`erl-on-load*`	Run expression when a namespace loads
`behaviour*`	Behaviour for the current namespace

Table 2. Clojerl Additional Special Forms

creating abstractions, which give the language its expressive power – protocols (Section 6), multimethods (Section 7), and macros (Section 8).

There are other important Clojerl aspects that are not discussed in this paper, such as the implementation of variadic functions and Clojerl's compiler. However, we consider these to be non-essential to understand the type of language Clojerl is and the principles it is built on. Further details on the inner-workings of specific Clojerl aspects will be addressed in future publications.

4 Namespaces & Vars

Every language needs to provide a way to organize the code written in them. Clojure does this through namespaces and vars and so does Clojerl, following the design principle of keeping as close to Clojure as possible.

A namespace holds any number of vars. Each var has a root binding which can be any value (e.g. an integer, a map or even a function). A var can also be instantiated, this means it is possible to get a runtime value that represents the var itself. This instantiated value holds data *about* the var (e.g. its documentation string), and is used extensively in the REPL (i.e. Clojerl's interactive shell, short for "Read, Eval, Print and Loop") to show information for namespaces and the vars they hold.

The BEAM provides modules as a way to structure code and group functions in a logical way. A compiled namespace in Clojerl is represented as a BEAM module. All information that might be needed during runtime, after the BEAM module is loaded, is kept in the module's attributes.

Each var in a Clojerl namespace is implemented with either one or two underlying functions in the BEAM module. When the var's root binding is a Clojerl function then the following two BEAM functions are created in the module:

1. An **implementation** function that executes the body of the Clojure function.
2. A **value** function which returns the instantiated var value.

For a var whose root binding is any value other than a function, a single function that returns this precise value is created in the BEAM module. As a way to illustrate this, Listing 1 shows a Clojerl namespace `f` followed by Listing 2, its equivalent (simplified) underlying representation in Erlang after it is compiled.

Listing 1. f.clje - A simple namespace

```
(ns f)
(def g 1)
(def (fn h [] 42))
```

Listing 2. f.erl - Erlang representation

```
-module(f).
-export([g__val/0, h__val/0, h/0]).
g__val() -> 1.
h__val() -> #{...}.
h() -> 42.
```

The fact that a namespace maps directly to a BEAM module, makes calling a Clojerl function from another BEAM language trivial. For example, calling the function `h` defined in Listing 1 from Erlang code would be as simple as the following: `f:h()`.

Additionally, it makes calling an Erlang function from Clojerl, indistinct from calling any other Clojerl function. A function call in Clojerl is expressed as follows:

```
(namespace/function arg1 arg2 ... argN).
```

Since Clojerl namespaces and BEAM modules are equivalent, `namespace` can be any Erlang module. For example, the expressions `(erlang/self)` and `erlang:self()` produce the same result – the identifier of the current process.

The disadvantage of bundling all vars in the same BEAM module is that the compilation process ends up being more complex. For example, Clojerl allows any var to be redefined through its interactive shell, which involves a recompilation of the namespace (i.e. BEAM module) where the var belongs. A recompilation of the namespace requires the compiler to keep a representation of the whole BEAM module, including the unmodified vars, so that only the redefined var can be replaced.

There are other complications during compilation that arise from mapping a namespace to a BEAM module than the one described above, but they are all encapsulated in the compiler. We make the trade-off of increasing the complexity in the compiler, in order to present a simpler model to the users of the language, i.e. a namespace is equivalent to a BEAM module.

An alternative implementation for namespaces and vars that we considered but discarded, is to create a BEAM module for each var in a namespace. The advantage of this approach is that it makes the compiling mechanism a lot easier, since every var's BEAM module can be compiled (and recompiled) in isolation from any other. The main disadvantage is that it makes calling a Clojerl function from Erlang a lot more complicated, as the name of a var's BEAM module would need to be unique and therefore some name mangling involved. For example, the Clojerl function `f/h` would map to

Data Structure	Literal Representation
List	`(+ 1 2 3)`
Vectors	`[:one 2 "three"]`
Map	`{:one 1, :two "two"}`
Set	`#{:one 2 "three"}`

Table 3. Clojerl Data Structures

a module named `clojerl_f_h` or similar. Another disadvantage is that given the BEAM's hard limit on the amount of atoms it can keep in memory, the creation of as many unique modules as vars, would considerably increase the number of atoms and the likelihood of surpassing this hard limit.

5 Data Types & Structures

Clojerl supports all of the data structures available in Clojure; however, only some of the data types are supported. There is a number of reasons behind this.

One reason for supporting all data structures is that Clojure is part of the Lisp family of languages, and as such its code is represented using the language's data structures (Table 3) – without them there is no language. Another reason is that Clojure's persistent immutable data structures are an essential part of its approach towards a better concurrency model, i.e. immutable values can be shared without having to worry about concurrent modifications. The BEAM includes some immutable data structures out of the box, which greatly simplify the implementation of Clojure's data structures in Clojerl (Section 5.2).

The reason behind supporting only some Clojure data types (Section 5.1), is that most of them are native data types provided by the host platform, e.g. JVM supports `int`, `float`, `long`, `double`, `byte`, `boolean`. For example, implementing support for double-precision floating-point in Clojerl would involve either a low-level implementation on the BEAM using C or a high-level implementation that most likely will result in a poor performance.

5.1 Data Types

Similar to some other Clojure implementations in different host platforms, such as ClojureScript, Clojerl takes the approach of using the native data types available in its own platform (BEAM) and maps these as best as possible to the ones in the canonical JVM implementation (Table 4). It is worth noting that the decision to map a `char` to an Erlang `binary()` with a single UTF-8 character, is to avoid the representation of single chars as simple integers, and also follows the same approach as ClojureScript.

Clojure also provides data types which are not native to the JVM platform, i.e. arbitrary precision integer, arbitrary precision decimal, ratio, keyword and symbol. Some of these are supported in Clojerl as User Defined Types (Section 5.3), others are mapped following the same principle as described above by other implementations (e.g. arbitrary precision integer), and some others are not supported at all (e.g. ratio). Table 4 shows how the Clojure specific literal data types are implemented in Clojerl.

JVM	BEAM
`boolean`	`boolean()`
`byte`, `short`, `int`, `long`	`integer()`
`float`, `double`	`float()`
`String`	`binary()` (UTF-8 encoded)
`char`	`binary()` (UTF-8 encoded)
Clojure	**Clojerl**
`BigDecimal`	Unsupported
`BigInt`	BEAM's `integer()`
`Keyword`	BEAM's `atom()`
`Nil`	BEAM's literal `undefined` atom
`Ratio`	Unsupported
`Regex`	User Defined Type
`Symbol`	User Defined Type

Table 4. Mapping JVM-to-BEAM and Clojure-to-Clojerl Data Types.

Clojure keywords are "symbolic identifiers that evaluate to themselves" [5]. Since their usage and properties are very close to Erlang's atoms, this is how they are represented in Clojerl.

Clojure symbols are "identifiers that are normally used to refer to something else" [5]. They are widely used in the language for different purposes and are implemented as a User Defined Type (5.3).

The available numeric operations in Clojerl are the ones the BEAM provides, and these only work with the numeric data types included in the platform. This is in line with Clojerl's *interoperability* design principle (Section 3). Clojure's `BigDecimal` and `Ratio` data types are therefore not included in Clojerl, because these types would not work with the BEAM's numeric operations. Implementing new operations to support both these types and the BEAM's types, would result in poor performance and sacrifice interoperability.

5.2 Data Structures

Clojerl implements all four main Clojure data structures (Table 3) plus all other Clojure secondary and auxiliary data structures, such as sorted maps and sorted sets. These data structures are all immutable and persistent by design, just like in Clojure.

The BEAM provides a set of immutable, persistent data structures (i.e. lists, tuples, and maps) that can be used for the purposes of representing their Clojerl counterparts. However, Clojerl's data structures have a crucial requirement which is not available in the BEAM: being able to attach metadata to them. Metadata in this context is defined as "a map of data

about the data structure" [5]. In Clojerl metadata is mostly used to signal the compiler about different aspects regarding the code, e.g. type hints are provided by adding a `:tag` entry in the metadata.

The *metadata* requirement means that it is not possible, for example, to use native BEAM lists as a drop-in replacement for Clojerl lists, as this would mean excluding a central feature for the list data structure. Therefore most Clojerl data structures are implemented as User Defined Types (Section 5.3) to enable both custom features (such as metadata support) as well extension through protocols (Section 6).

5.3 User Defined Types

User defined types can be created in Clojerl using the `deftype` macro (also present in Clojure) as shown in Listing 3.

Listing 3. Defining a type `Person`

```
(deftype Person [first-name last-name])
```

Clojerl represents the value of a User Defined Type as a tagged Erlang map. This means the map contains a special key (i.e. Erlang atom `__type__`) with the name of the type as an associated value. The fields defined for the type are added as extra key-value entries in the map. Listing 4 shows the Erlang map representation of a value of the `Person` type, where the value for `first-name` is "Jane" and for `last-name` is "Doe".

Listing 4. Map representation of a User Defined Type value

```
#{ '__type__'   => 'Person'
 , 'first-name' => <<"Jane">>
 , 'last-name'  => <<"Doe">>
 }
```

As discussed in Section 5.2, Clojerl data structures are implemented as user defined types to be able to support the addition of metadata. Listing 5 shows the type specification for a Clojerl list illustrating how metadata is kept for it. The `items` entry holds the elements in the list, and the `metadata` entry holds any value representing its metadata. The same approach applies to all user defined types that have metadata associated with them.

Listing 5. Erlang type spec for a Clojerl list

```
-type type() ::
  #{ '__type__' => 'clojerl.List'
   , items      => list()
   , meta       => undefined | any()
   }.
```

An alternative representation of user defined types in Clojerl that was considered uses Erlang records, defined as shown in Listing 6. The record's `name` field contains the type's name, `data` holds the most important underlying information and `info` keeps additional values that are not part of the type's value, e.g. metadata. Since Erlang records are just tagged tuples, the idea is that any tuple with the special `?TYPE` tag, is a value of a user defined type whose name is in the `name` field.

Listing 6. Record representation for a User Defined Type

```
-defined(TYPE, '__type__').
-record(?TYPE, { name = ?MODULE :: atom()
               , data           :: any()
               , info = #{}     :: map()
               }).
```

However, the record representation has two main shortcomings. First, it results in code that is hard to read. And second, any changes to the record (e.g. adding a new field) triggers a lot of changes in other parts of the codebase, which are not relevant or necessary in most places. Therefore we considered and explored the approach of using tagged maps instead.

Clojerl uses maps as the representation because they are less obscure than records. Maps are open for extension as new fields can be added as needed, which is more aligned with Clojure's philosophy.

Usage of maps for user defined types produces code that is readable and simpler to maintain. Whereas having a fixed record for all user defined types, would result in occasionally having a complex internal representation in the `data` field, that would then need to be pattern matched each time a value is retrieved or set.

The downside of using maps is that operations on these demonstrate worse performance than on records (i.e. tuples). Operations of creating a new tuple and accessing an element in a tuple are generally faster than the equivalent operations using a map. The results of a simple benchmark we performed[1] showed that some operations were up to 40% slower.

Clojerl makes the conscious choice of using maps. Even though maps perform worse than records for certain operations (e.g. creating a new record is faster than creating a new map), they provide a simpler and more extensible representation for user defined types.

6 Protocols

Like in Clojure, applications and libraries in Clojerl are written in terms of abstractions [5]. However, while Clojure's language abstractions are defined mainly by Java interfaces, Clojerl defines its abstractions exclusively through protocols.

Support for protocols in Clojerl slightly deviates from what is provided by Clojure, due to the BEAM not offering any kind of type hierarchy that is available in other platforms, such as JVM and Javascript V8 engine. Therefore it is not possible in Clojerl to implement a protocol for a type and have that implementation automatically applicable for a group of derived types.

[1] https://github.com/clojerl/clojerl/issues/364

Importantly, this does not reduce the power of protocols, since their main goal is still fulfilled: solving the expression problem [13], i.e. allowing to "[...] define a datatype by cases, where one can add new cases to the datatype and new functions over the datatype, without recompiling existing code [...]". Protocols open up the extension of possible closed data types that the user might not own.

A protocol in Clojerl is defined similarly to Clojure (Listing 7). The implementation of each protocol is done through a BEAM module that exports the functions defined in the protocol, plus two more helper functions:

1. `'__satisfies?__'/1`: determines if a value satisfies the protocol.
2. `'__extends?__'/1`: determines if a type extends the protocol.

Listing 7. Defining a protocol `IFoo`

```
(defprotocol IFoo
  (bar [x] "Description"))
```

The body of the functions defined in the protocol contain the *dispatch logic* necessary to find the correct implementation function, if one is available. The dispatch logic is equivalent to the Erlang `case` expression in Listing 8, where the value in the case expression is the type of the first argument. Each case clause matches on specific types that implement the protocol. When the type does not implement the protocol a default clause throws an error informing the user there is no such implementation.

Listing 8. Dispatch logic for function `bar` from protocol `IFoo`.

```
bar(X) ->
  case type(X) of
    'clojerl.List' -> 'clojerl.List':bar(X)
      ;
    %% ...
    _ -> error("Type does not implement 
      protocol")
  end.
```

A protocol can be extended either when a new type is created (Listing 9) or by implementing the protocol for an existing type (Listing 10).

Listing 9. Implement protocol for new type

```
(deftype MyType [x]
  IFoo
  (bar [_] "Implementation for MyType"))
```

As discussed in Section 5 every type is backed by a BEAM module and, additionally, all types are open for extension. As a result, every time a protocol is implemented for a type, two things need to happen.

T-1: The dispatch logic described above needs to be updated.

T-2: The functions that implement the protocol need be added to an existing or new module.

Clojerl accomplishes T1 by updating the BEAM module for the protocol with a new clause in the dispatching case expression. And T2 is accomplished either by creating a new protocol-type specific BEAM module, if the type already existed; or by adding the implementation function in the module that backs the type, if it is a new type that is being created.

Listing 10. Implement protocol for existing type

```
(extend-type clojerl.Integer
  IFoo
  (bar [_] "Implementation for Integer"))
;; or
(extend-protocol IFoo
  clojerl.Integer
  (bar [_] "Implementation for Integer"))
```

The reason for creating a new protocol-type specific BEAM module when the type exists, comes from the fact that when a module is recompiled and loaded, the previous version needs to be purged and deleted. This involves killing all BEAM processes that might have been using the code from the deleted version. This is a side-effect that can bring unintended consequences, which is what the creation of a separate protocol-type specific BEAM module tries to avoid.

7 Multimethods

Clojerl supports multimethods in the same way as Clojure. Multimethods provide runtime polymorphism and are a tool to create abstractions, just like protocols. The main difference is that multimethods perform the dispatching based on the value returned by a user-defined function, whereas protocols do so only based on the type of the first argument.

A multimethod is initially defined by providing a name and a dispatching function, as show in Listing 11. The user then needs to specify implementations for the multimethod, these are referred to as its methods. Each method consists of a dispatch value and the code to be executed for that value. It is possible to register multiple methods for a multimethod either at compile time or at runtime.

Listing 11. Defining a multimethod and registering methods.

```
(defmulti foo keyword)
(defmethod foo :bar [x] :do-bar)
(defmethod foo :baz [x] :do-baz)
```

A multimethod is used like a regular function. When a multimethod is called, its dispatching function will be applied to the multimethod's arguments to produce a dispatching value, which will then be used to match one of the registered methods. This mechanism is illustrated by the Erlang code shown in Listing 12.

Listing 12. Equivalent Erlang code for multimethod foo.

```
foo(X) ->
  Val = keyword(X),
  DispatchMap =
    #{ bar => fun (X) -> 'do-bar' end,
     , baz => fun (X) -> 'do-baz' end
     },
  Fun = maps:get(Val, DispatchMap),
  Fun(X).
```

The implementation of multimethods in Clojerl differs from Listing 12 in that it keeps the dispatch map in a separate module. This is due to adding entries (i.e. methods) at runtime would trigger a modification and reloading of the module containing the map. Any process that uses the module at that time would be killed. We minimize the likelihood of a process dying by keeping the dispatch map in a separate module. This module contains a single function whose return value is the multimethod's dispatch map.

8 Macros

Macros are an essential feature of Clojure. A big part of the language are the macros provided in the standard library that allow to build idiomatic expressions. For that reason, Clojerl provides the same support for macros as Clojure.

Macros are arguably also an essential part of any Lisp language, not just Clojure. They are a way to transform code at compile time, which is somewhat similar to Erlang's parse transforms. But macros in a Lisp language bring even more power to the user, since the representation of the code that is being transformed is the same as the code that is written: lists. In the case of Clojerl and Clojure, there's not only lists used in code, but also the other data structures presented in Table 3 (e.g. vectors and maps).

By using macros the user can define new syntactic constructs that adapt better to the problem they are trying to solve, effectively creating a Domain Specific Language.

Most languages from the Lisp family implement macros in a similar way: as a function evaluated at compile time whose result is fed into the compiler. Clojure and Clojerl are no exception. The implementation of macros in Clojerl is very close to the one in Clojure, where macros differ from regular functions in two aspects:

1. The var associated to the macro contains a `:macro` tag in its metadata, to signal the compiler this is a macro.
2. The macros always receives at least two arguments:
   a. `&form`: the actual expression (as data) that is being invoked.
   b. `&env`: a map of local bindings at the point of macro expansion.

To better illustrate the power of macros consider `defmacro`, which is provided in Clojerl to define new macros. `defmacro` is itself a macro defined in the standard library through the use of metadata (Section 5.2) and the special forms `def` and `fn` (Table 1). Listing 13 presents an extract from the definition of `defmacro`. It shows how the two special forms are combined and also the usage of metadata to signal the compiler a macro is being defined.

Listing 13. Definition of `defmacro`

```
(def ^{:macro true}
  defmacro
  (fn [&form &env name & args]
    ;; removed code
    ))
```

9 Evaluation

In this section we evaluate the performance of Clojerl by comparing the execution time of its expressions with the corresponding expressions of Clojure (in the JVM). The expressions we chose as points of the comparison, are ones we consider either fundamental to the language itself or heavily used in practice. The full list of expressions that we evaluated to investigate the performance of Clojerl is presented in Table 5. In this paper we discuss a representative subset of these expressions.

A) Simple function call (Section 9.2)
B) Dynamic function application (Section 9.3)
C) Get last item in a Range (Section 9.4)

Further details of the experiments from Table 5 are available at https://github.com/clojerl/benchmarks. The experiments are run using the same code base for both Clojerl (using the BEAM) and Clojure (using the JVM). This is possible due to Clojerl's guiding principle of allowing portability between the two languages.

9.1 Configuration & Setup

Table 6 shows the hardware and software configurations used in the experiments. A script starts the execution of each experiment, first for Clojure using the Leiningen [2] build tool (the default tool for the language) and then for Clojerl using the `rebar3_clojerl` [3] plugin.

Each experiment's expression is run a 100 times before collecting the samples for the individual runs, to warm up the VM. This is mostly necessary for Clojure on the JVM which includes JIT compilation, but it also benefits the execution time in Clojerl to enable the loading of all required modules.

Each experiment runs the expression 10,000 times and records the time it takes each individual run to complete. The time is measured using `erlang:monotonic_time/1` in Clojerl and `System.nanoTime()` in Clojure. Both functions return the results in nanoseconds (ns).

The number of collected samples (i.e. 10,000) is chosen to make sure the expression runs for a considerable amount of time, and also to get a sample size big enough to extract

[2] https://leiningen.org/
[3] https://github.com/clojerl/rebar3_clojerl

#	Experiment	Units	Clojure Mean	Clojure Std. Dev.	Clojure Median	Clojerl Mean	Clojerl Std. Dev.	Clojerl Median
0.	*No expression (measurement cost)*	ns	70.0063	3.7815	69.0000	120.7284	97.4033	118.0000
1.	Constant expression	ns	7.2791	161.5242	4.0000	7.7721	172.8567	5.0000
2. ✓	Simple function call	ns	94.9017	193.0736	87.0000	17.4028	255.3728	10.0000
3.	List creation	ns	503.3183	632.7734	346.0000	99.7628	357.3604	91.0000
4.	Protocol dispatch	ns	106.3395	172.7034	95.0000	77.8360	243.7750	70.0000
5.	Read expression from string	µs	11.2877	2.8173	11.6320	76.2471	22.1090	74.2530
6.	Tight loop	µs	93.9456	28.3197	93.4420	1513.17	30.9575	1507.98
7. ✓	Dynamic function application	ms	38.7237	1.6787	38.1172	102.3978	0.7210	102.0682
8. ✓	Last item in range	ms	50.1421	0.5774	50.0696	463.8561	2.9287	463.5184

Table 5. List of Evaluated Expressions.

Hardware	
Vendor	Dell Inc.
Model	PowerEdge C6220 II
Memory	62 GiB
CPU	Intel(R) Xeon(R) CPU E5-2640 v2 @ 2.00GHz
Software	
Operating System	Ubuntu 18.04.2 LTS
Java	OpenJDK 1.8 (build 25.212-b03)
Clojure	1.10.1
Erlang/OTP	21.1
Clojerl	0.6.0

Table 6. Hardware and Software Configuration

statistics. The large number of samples means that it is very likely some will be outliers. This is because other factors come into play when running a program for a period of time, e.g. garbage collection, processes at the operating system and platform level that use the same resources such as memory, CPU, and I/O. These other processes cannot be eliminated since they are necessary for the operating system and the platform to be able to run.

To eliminate the measurement cost introduced by the recording of the measurements, we ran the "No Expression" experiment (Table 5). The experiment collects each sample by capturing two consecutive timestamps with no expression in between the captures. We estimate the measurement cost as the lower quartile of the samples – a conservative compromise between the "median" and the "minimum" values. This estimated measurement cost is already subtracted from the values shown in Table 5 (for experiments 1-8), as well as the figures shown in the experiment sections that follow.

9.2 Experiment-A: Simple Function Call

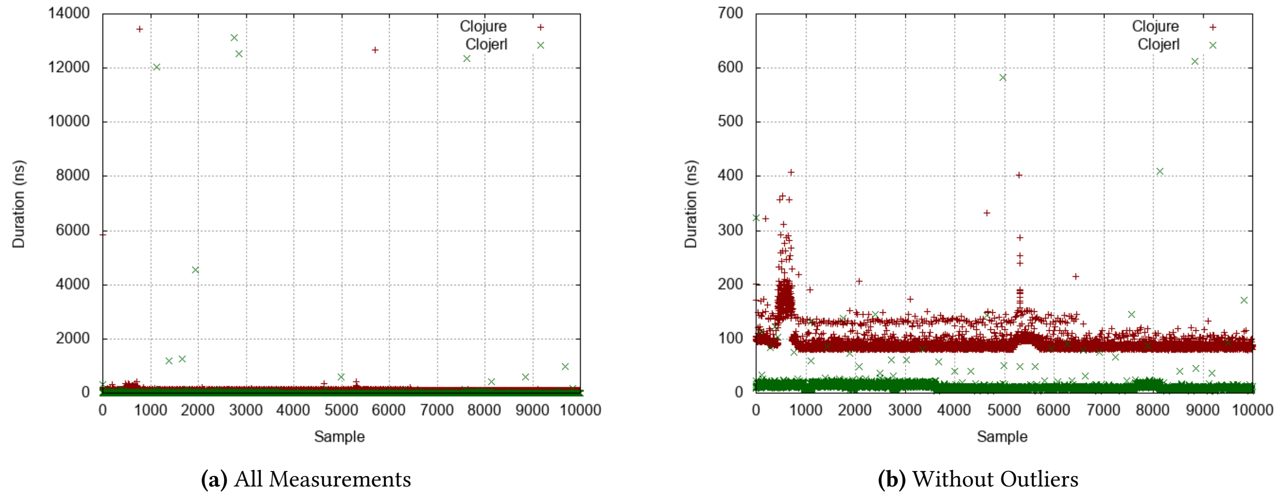
This experiment aims to analyze the performance of a simple function call, `identity`, which only returns the provided argument without doing any operations on it (Listing 14). Function calls are an essential tool in any functional programming language. Therefore it is important for their execution to be as fast as possible.

Listing 14. Simple Function Call

```
(identity 1)
```

Results in Table 5 (experiment #2) show that *a simple function call is approximately five times faster in Clojerl than in Clojure*, i.e. mean times are 17.4ns and 94.9ns respectively. We attribute this to Clojerl's `identity` call being compiled down to an equivalent Erlang function call; that is the BEAM considers functions as first-class citizens and a function call gets compiled to a low-level virtual machine assembly instruction. The JVM offers low-level instructions for function calls as well; however, Clojure on the JVM introduces one extra level of indirection for a function call. Since `identity` is a var (Section 4) whose root binding is a function, performing a function call involves the following two operations: (1) fetching the value of the root binding (i.e. an instance of the function) and (2) making the call to the function. This level of indirection for a function call in Clojure is likely to be the root cause for the time penalty we see in the results.

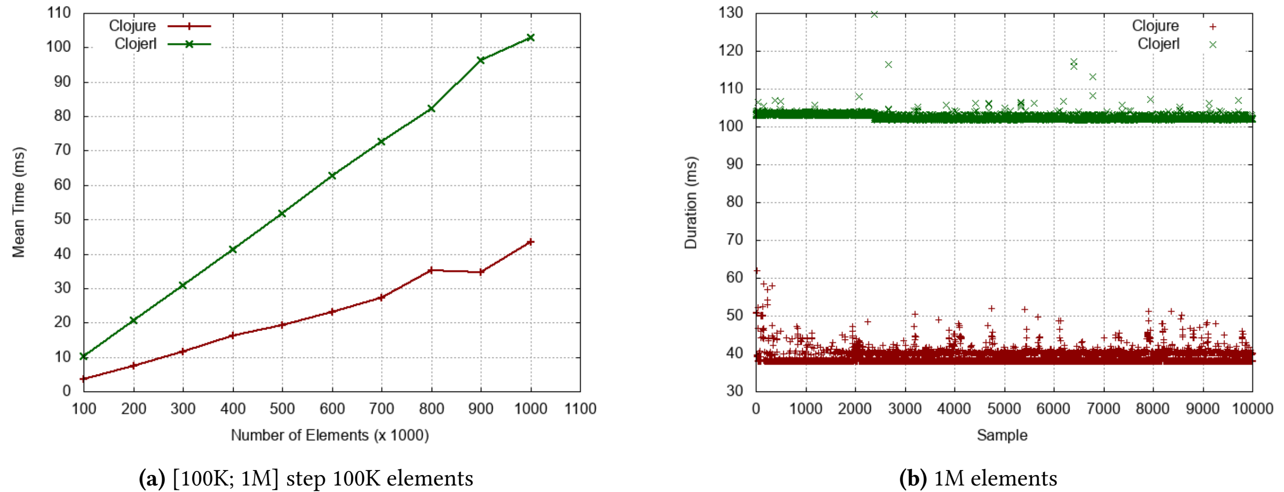
Figure 1a shows a number of outlier spikes in both Clojerl and Clojure samples. In both cases they happen throughout the experiment; therefore, they are very likely to be neither language nor VM specific, but caused by external factors. Since these outliers overshadow the details around the samples' mean in both Clojure and Clojerl, we include Figure 1b which shows the same data as Figure 1a but without the outliers. The criteria to identify a value as an outlier is whether it is over 3 standard deviations above the mean. Using this criteria three outliers are removed from Clojure's samples and eight outliers from Clojerl's, out of a total of 10,000 samples in each case.



(a) All Measurements

(b) Without Outliers

Figure 1. Simple Function Call



(a) [100K; 1M] step 100K elements

(b) 1M elements

Figure 2. Dynamic Function Application.

9.3 Experiment-B: Dynamic Function Application

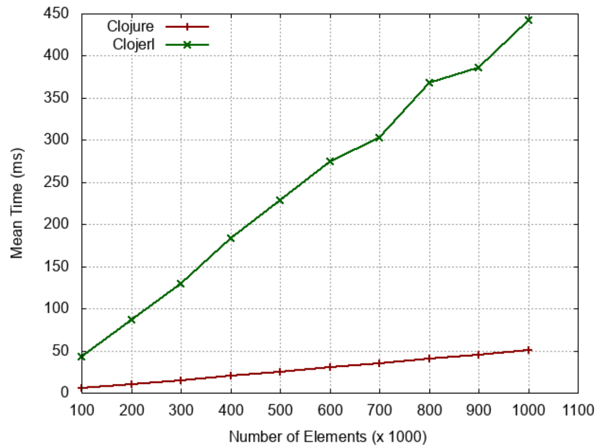
In this experiment we analyse the performance of applying a list of a large number of arguments – between 100,000 and 1,000,000 elements – to a function. Listing 15 illustrates the application of a function with a list of 1,000,000 elements. The outer `let` expression is not included in the measurement, only the inner `(apply + x)` expression is sampled.

Listing 15. Dynamic function application

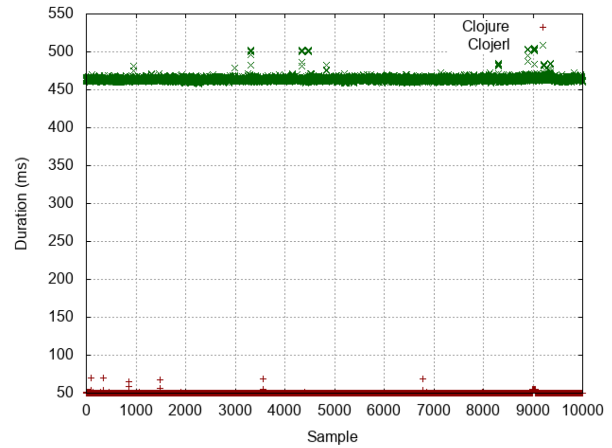
```
(let [x (into [] (range 1000000))]
  (apply + x))
```

Functions are first-class citizens in Clojerl and Clojure. Higher-order functions are very common, i.e. functions that take other functions as arguments. The functions in Clojure's standard library (i.e. `clojure.core` namespace) make heavy use of `apply` to call a function provided as an argument. For example, the frequently used `map` function takes a function `f` and a list of values `xs` and returns another list where `f` has been applied to each value in `xs`.

The results for a dynamic application of a large list of arguments to a function are presented in Figure 2a. The mean time is calculated by collecting 10,000 samples for each number of elements from 100,000 to 1,000,000 with a step of 100,000 elements. The results show that for `[100K; 1M]` elements Clojerl follows the `y=42.7x-1.33e6` trajectory, while Clojure follows the `y=105x-0.48e6` trajectory. For the experiment with 1,000,000 elements the mean time for Clojerl is 102.4ms, which is approx. 2.5 times more than for the corresponding Clojure results (38.7ms) as shown in Table 5 (experiment #7).



(a) [100K; 1M] step 100K elements



(b) 1M elements

Figure 3. Get Last Item in a Range.

We attribute Clojure's superior performance in this experiment to the combination of the following two factors: (1) the use of a specialized implementation for variadic vs. non-variadic functions and (2) the availability of fast runtime dispatching using JVM's interfaces. Both these features are currently absent in Clojerl.

That is, a dynamic function application in Clojerl requires some runtime processing to resolve the arity of the function that should be called. In Clojerl a single function can have different arities, which in the underlying BEAM representation translates to different functions each with its own arity. This should not present an issue, but the runtime processing is more involved because a Clojerl function can be defined to accept an *indefinite number of arguments* as well. This is referred to as "variadic arity". In Clojerl the logic used for resolving which arity should be used at runtime is the same for *all* functions (variadic and non-variadic).

Clojure also supports defining variadic arity functions and needs to perform some runtime processing to resolve which arity to use as well. The resolution for a dynamic function application in Clojure makes use of inheritance and interfaces. Inheritance is used by having a specific implementation (i.e. class) for functions that include a variadic arity and functions that do not. Interfaces are used to abstract away the implementation and call the method with the arity that is being resolved.

The information whether a function is variadic or not is available at compile-time in Clojerl as well. This should make it possible for Clojerl to use a similar approach as Clojure in the future to improve its performance.

As shown in Figure 2b both implementations present some outliers. Even though some of them are 3 standard deviations above the mean, none overshadow the values around the mean as in Experiment-A (Section 9.2). The Clojure samples show noisier spikes up until approximately the 500th sample, and then present a consistent and similar behaviour as the Clojerl samples. This initial noise is likely due to JIT compilation taking place in the JVM during the first few Clojure samples.

9.4 Experiment-C: Get Last Item in a Range

In this experiment we analyze the performance of the function `last` when applied to a range, i.e. getting the last item of a range. Listing 16 illustrates the expression with the range of 1,000,000 elements. The outer `let` expression is not included in the measurement, only the inner `(last r)` expression is sampled.

Listing 16. Last item in Range

```
(let [r (range 1000000)]
  (last r))
```

A range is a sequence of numbers that is generated by specifying its bounds (start and end) and a step to use between each consecutive values. Ranges are mostly used in expressions like `for` (i.e. Clojerl's list comprehensions construct) or when there is a need to generate the indexes for a collection. These situations are quite common when using both Clojerl and Clojure.

To evaluate the performance of the `(last r)` expression we ran the experiment with the range `r` containing a number of elements from 100,000 to 1,000,000 with a step 100,000 elements (Figure 3a). The mean time is calculated by collecting 10,000 samples for each number of elements. The results show that for the range `[100K; 1M]` Clojerl follows the `y=50.2x+7727` trajectory, while Clojure follows the `y=442x+1.37e6` trajectory. For a range of 1,000,000 elements, Clojerl's mean time of 463.8ms is almost an order of magnitude larger than Clojure's mean of 50.1ms (Table 5, experiment #8).

The optimizations done in Clojure's `range` *implementation are very likely to be the main cause of the large difference in the execution times we see in this experiment.*

Getting the last element of a range involves going over each element until we get to the last one. Ranges in Clojure are implemented by pre-calculating the elements of the range in chunks, generating each chunk as needed. In contrast, the Clojerl implementation is more naive, since it checks the bounds for the range each time the next element in the range is needed, therefore doing more work for each item. This extra work adds up when dealing with ranges that contain a large number of elements, as the results from this experiment show.

It should be possible to apply the same optimization techniques to Clojerl's range implementation, and thus reduce the overhead when iterating through the items in a range.

There are some outliers that can be observed in Figure 3b, both in Clojerl and Clojure. At first sight Clojerl seems to present more outliers than Clojure. However, if we count values above 3 standard deviations of the mean, we find that the number of outliers is similar: 61 in Clojerl and 69 in Clojure. This is due to the fact that even though Clojerl's outliers are further away from the mean, its standard deviation is also larger than Clojure's in this experiment.

10 Conclusion

We have presented Clojerl – a new language that aims to combine the expressive power of Clojure with the high availability and massive scalability of the BEAM. Clojerl is designed to be as close to Clojure as possible, while at the same time enabling interoperability with the BEAM. This has two important benefits: (1) Clojure code can be easily ported to Clojerl (and vice versa) and (2) other BEAM languages can easily interact with Clojerl (and vice versa).

Clojerl provides a small but powerful set of data types and data structures (Section 5). Due to all data structures being built on common abstractions, they can be manipulated and combined using the same functions included in the `clojure.core` standard library. The language also provides a way of creating user defined types (Section 5.3), which combined with protocols (Section 6) can be used to create new abstractions or extend existing ones.

Clojerl offers a mechanism for runtime polymorphism through its multimethods (Section 7). These allow the user to define a custom dispatching function, along with a set of expected values and the code to run for each of them. Multimethods can be also extended at runtime, which makes them more flexible and versatile than Erlang's pattern-matching.

Like all languages from the Lisp family, Clojerl supports macros (Section 8). Macros are essentially functions that get evaluated at compile-time and allow extending the syntax of Clojerl itself, effectively enabling the creation of a domain specific language, catered to the problem being solved.

By using the same code for Clojure and Clojerl when running the experiments in Section 9, we have been able to exemplify the value of our design principle of keeping Clojerl and Clojure as close as possible (i.e. to facilitate portability). While the fact that *basic Clojerl's expressions outperform Clojure* (experiments #1-#4 in Table 5), provides reassurance that by applying optimisation techniques, we will be able to significantly improve the performance of more complex expressions.

Further experiments are required to identify other areas where Clojerl could improve its performance, and where it already outperforms Clojure. Multimethods are a likely candidate for Clojerl's improvements, since they are an important and versatile language feature that has been heavily optimised in Clojure.

It is currently possible to use any OTP tool by relying on Clojerl's interoperability features, but the user experience is not frictionless. We plan to extend the language to provide support for all OTP abstractions (e.g. `gen_server`), making Clojerl a better equipped language to build OTP applications.

The results from our experiments strongly suggest that Clojerl has a potential to provide a competitive performance while offering the rich set of programming language features available in Clojure.

Acknowledgments

We would like to thank the School of Computing at Glasgow University, UK for providing access to their GPG cluster.

Juan would like to thank his spouse Catalina for all her love and support.

References

[1] 2020. *Erlang Run-Time System Application (ERTS) - User's Guide.* https://erlang.org/doc/apps/erts/absform.html
[2] Joe Armstrong. 2013. *Programming Erlang: software for a concurrent world.* Pragmatic Bookshelf.
[3] Richard Carlsson. 2001. An introduction to Core Erlang. In *Proceedings of the PLI*, Vol. 1. Citeseer.
[4] Michael Fogus and Chris Houser. 2011. *The joy of Clojure.* Manning.
[5] Rich Hickey. [n.d.]. *Clojure Documentation.* https://clojure.org/
[6] Mark McGranaghan. 2011. Clojurescript: Functional programming for javascript platforms. *IEEE Internet Computing* 15, 6 (2011), 97–102.
[7] Eric Merritt. [n.d.]. *Joxa: a concurrent distributed Lisp.* http://joxa.org/
[8] David Miller. [n.d.]. Clojure CLR.
[9] Jeremy Singer. 2003. JVM versus CLR: a comparative study. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java.* 167–169.
[10] Erik Stenman. 2017. *The BEAM Book.* Self-published.
[11] Dave Thomas. 2018. *Programming Elixir => 1.6: Functional|> Concurrent|> Pragmatic|> Fun.* Pragmatic Bookshelf.
[12] Stefan Tilkov and Steve Vinoski. 2010. Node. js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing* 14, 6 (2010), 80–83.
[13] Mads Torgersen. 2004. The expression problem revisited. In *European Conference on Object-Oriented Programming.* Springer, 123–146.
[14] Luke VanderHart, Stuart Sierra, and Christophe Grand. 2010. *Practical Clojure.* Vol. 232. Springer.
[15] Robert Virding. [n.d.]. *List Flavoured Erlang.* http://lfe.io/