A History of Clojure

RICH HICKEY, Cognitect, Inc., USA

Shepherd: Mira Mezini, Technische Universität Darmstadt, Germany

Clojure was designed to be a general-purpose, practical functional language, suitable for use by professionals wherever its host language, e.g., Java, would be. Initially designed in 2005 and released in 2007, Clojure is a dialect of Lisp, but is not a direct descendant of any prior Lisp. It complements programming with pure functions of immutable data with concurrency-safe state management constructs that support writing correct multithreaded programs without the complexity of mutex locks.

Clojure is intentionally hosted, in that it compiles to and runs on the runtime of another language, such as the JVM. This is more than an implementation strategy; numerous features ensure that programs written in Clojure can leverage and interoperate with the libraries of the host language directly and efficiently.

In spite of combining two (at the time) rather unpopular ideas, functional programming and Lisp, Clojure has since seen adoption in industries as diverse as finance, climate science, retail, databases, analytics, publishing, healthcare, advertising and genomics, and by consultancies and startups worldwide, much to the career-altering surprise of its author.

Most of the ideas in Clojure were not novel, but their combination puts Clojure in a unique spot in language design (functional, hosted, Lisp). This paper recounts the motivation behind the initial development of Clojure and the rationale for various design decisions and language constructs. It then covers its evolution subsequent to release and adoption.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: Clojure, Lisp

ACM Reference Format:
Rich Hickey. 2020. A History of Clojure. *Proc. ACM Program. Lang.* 4, HOPL, Article 71 (June 2020), 46 pages.
https://doi.org/10.1145/3386321

Contents

Author's address: Rich Hickey, Cognitect, Inc., USA, richhickey@gmail.com.

2475-1421/2020/6-ART71
https://doi.org/10.1145/3386321

1 INTRODUCTION

The objective for Clojure can be summarized most succinctly as: I wanted a language as acceptable as Java or C#, but supporting a much simpler programming model, to use for the kinds of information system development I had been doing professionally.

I started working on Clojure in 2005, during a sabbatical I funded out of retirement savings. The purpose of the sabbatical was to give myself the opportunity to work on whatever I found interesting, without regard to outcome, commercial viability or the opinions of others. One might say these are prerequisites for working on Lisps or functional languages. I budgeted for two years of self-directed work, and Clojure was one of two projects I pursued. After about a year I decided the other project (a cochlear modeling and machine listening problem) was more of a research endeavor that might require two to five more years, so I dedicated myself at that point to getting Clojure to a useful state. I announced and released the first version of Clojure, as an open source project, in the fall of 2007. I did 100% of the implementation work during this time, and more than 90% through the 1.2 release in 2010. Subsequent to release Clojure benefited greatly from the feedback, suggestions and effort of its community. I am accepted by the community as "benevolent dictator for life" (BDFL) and continue to make all decisions relating to its evolution. Clojure is full of the great ideas of others, but I alone take responsibility for its faults.

2 BACKGROUND AND MOTIVATION

2.1 Tuesday Afternoon

I had been doing commercial software development since 1987, almost always as the development lead and primary architect, first in C++, then Java and C#, as was common then and, in testament to institutional inertia, is still now thirty years later. I worked on scheduling systems, broadcast automation, yield management, audio recognition, exit poll tabulation and election projection et al. I would broadly characterize my work, and the work most commonly done by professional programmers, as *information systems* programming. Most developers are primarily engaged in making systems that acquire, extract, transform, maintain, analyze, transmit and render information—facts about the world. Most often, this information documents some human activity, be that of customers, suppliers, advertisers, travelers, voters, members, students, patients etc and must deal with all the irregularity thereof. This is in stark contrast to *artificial systems*, e.g., programming language compilers, which make up their own rules, in fully enumerated spaces, can eliminate irregularity and can reject anything which does not conform.

Information system programmers have the thankless task of attempting to superimpose somewhat regular models over information and real-world activity that refuses to comply. For instance, in music scheduling, trying to decide: whether artists have songs or songs have artists. Or optimizing a scheduler to spread out the plays of songs by each artist, except on 'twofer Tuesdays' when songs by each artist must be played in adjacent pairs. In information systems programming, twofer Tuesdays are everywhere.

2.2 I'm a Believer

By the mid 1990's I was a C++ expert, taught advanced C++ as an adjunct at NYU, and was a proponent and advocate of the benefits of static typing (but neglected the tradeoffs, sorry students!). I was happily discovering type parameterization tricks [Hickey 1996], running a const-correct shop etc.

However, over time, in my experience, the suitability-to-task of these statically typed class models for information systems programming was quite low, and the benefits of the type checking minimal, especially in addressing the number one actual problem faced by programmers: the overwhelming

complexity inherent in imperative, stateful programming. As programs grew large, they required increasingly Herculean efforts to change while maintaining all of the presumptions around state and relationships, never mind dealing with race conditions as concurrency was increasingly in play. And we faced encroaching, and eventually crippling, coupling, and huge codebases, due directly to *specificity* (best-practice encapsulation, abstraction, and parameterization notwithstanding). C++ builds of over an hour were common.

In the years immediately preceding work on Clojure I worked on the system to be used for the national exit poll in the U.S. This system involved automating large statistical models. Early on we decided that an imperative approach to doing the stats was a misfit, being not mathematical and also due to the expected concurrency throughout the system. After exploring F# for the task (and finding it insufficiently expressive), we decided to code the stats in C# like the rest of the system (this was a Microsoft .Net shop). I designed some custom immutable data structures whose use would be co-aligned with the figures in the statistical specifications (so statisticians who did not know C# could look at the code and understand it), and a functional library for manipulating the structures and doing the calculations. This was a great success, yielding much simpler code that we did not, and still do not, worry about. However, the resulting C# code was, in the eyes of both new and experienced C# developers, bizarre and non-idiomatic. Thus functional programming was a win, but FP in a non-FP language was not something I could see being widely applied.

2.3 Who Do You Love?

I had always been a language geek, playing with Lisp, Prolog and Smalltalk in my spare time. When I became an independent consultant in 2000, I spent more time with Common Lisp [Steele Jr et al. 1990] and wrote a couple of real systems in it. It was a revelation. Huge layers of unnecessary complexity simply vanished. I had the flexibility to use exactly as much language as was needed for the problem. The percentage of code directly related to the domain increased. Development was much faster, the resulting program was more general and easier to change. It was impossible to avoid the sinking feeling that I had been "doing it wrong" by using C++/Java/C# my whole career. I needed another choice more like Common Lisp or I wouldn't be able to continue as a professional software developer.

While language features matter, the primary hurdle to language adoption by professionals is acceptability to developers and stakeholders. Thus, Clojure did have to be at minimum *practical* for developers I would work with, and companies I might work for, or else I couldn't use it to make a living. I took this as an agenda item for its design but not as motivation to seek their input in advance, because at the time the advice from professional developers about Lisp was "it's dead" and about functional programming was "what's that?", and about writing a new programming language was "you're crazy".

The year 2005 may have represented the nadir of language diversity in commercial software development, with ISVs primarily using C/C++ and many businesses considering themselves either Java or .Net 'shops'. This was before the ascent of dynamic languages for web development (e.g., Ruby on Rails), and many companies were reluctant to deploy and operate anything that didn't run on the JVM or CLR. I did commercial work in Common Lisp twice - a scheduling system and a yield management system. The first time the program had to be rewritten in C++ in order to be acceptable for deployment, an arduous task (though I was a seasoned C++ programmer) that took longer than the initial development, was much more code, and did not run significantly faster. In the second case again Common Lisp was not acceptable for deployment, so I designed the program to generate SQL stored procedures for delivery to the client and runtime execution.

Prior to embarking on Clojure in 2005, I had made several attempts to create a bridge between the JVM and Common Lisp, in order to increase the latter's practicality and acceptability. These

were DotLisp [Hickey 2003], an interpreted Lisp with host interop for the CLR, jFli [Hickey 2004], a library that embedded a JVM in CL, and Foil [Hickey and Thorsen 2005], a library that exposed a similar API but used IPC between the CL runtime and the JVM. None of these yielded production-level solutions, but they definitely informed Clojure's ultimate host syntax and fed the idea of Clojure being *hosted*.

2.4 Saturday Night's All Right...

I attended the Lightweight Languages Workshop at MIT in 2002 and 2003 (so the bug had alighted). After one of the conferences there was pizza, and I sat with two programming language researchers (whose organizations I will not mention and names I do not recall). They were speaking with derision about how one of their colleagues had gotten involved with databases. "I don't think I've ever written a program that used a database" said one, "Neither have I!" said the other, and they chuckled with great self-satisfaction. I was aghast, having never written a commercial program that did not involve a database.

As an application architect, working with databases taught me several important things about programming languages and their role in systems. The primary information model should not live in any application program. The more elaborate the type abstractions built up inside the programming language the less likely they would be shared by any other system participant, and the more involved and necessary marshaling would be to get in/out of the program's model. The durable information would outlive any program both in the small (execution lifecycle) and the large (utility over time). Programing language constructs are not at the center of system designs.

At some point I had my "I'm mad as hell and I'm not going to take this anymore!" moment, in my backyard, on my hammock.

3 INITIAL DESIGN AND LANGUAGE FEATURES, WITH RATIONALE, 2005-2007

I spent the first two years of work on Clojure, prior to its public availability, forming a rationale, doing design and implementing the initial feature set. I consider forming a rationale a significant part of the work of making Clojure. For narrative completeness I have included a few features that followed this time period, and noted their timing when doing so.

3.1 Clojure, the Lisp

Clojure is a Lisp, but what does that mean? Lisp is a fascinating set of ideas, a family of languages without a monotonic lineage [Steele and Gabriel 1996]. There are some shared characteristics, all of which I considered positive attributes to retain for Clojure: code is data; independent read/print; small core; runtime tangibility; the REPL; and extreme flexibility.

3.1.1 Code is Data. A Lisp program is defined in terms of the interpretation of its core data structures, rather than by a syntax over characters. This makes it easy for programs to be a source of programs and for program transformations to be functions of data → data, and in both cases for the standard library to be of substantial utility. The conversion of character streams into data is handled by separate functions of the reader.

While much is made of these facilities as a recipe for *syntactic abstraction* (e.g., macros), syntactic abstraction via code-is-data is just a single exemplar of the value of defining any process as functions of data → data wherein the standard data structures and library are of significant utility and reusability. This is in stark contrast to the common tendency to design a bespoke interface and types for every problem.

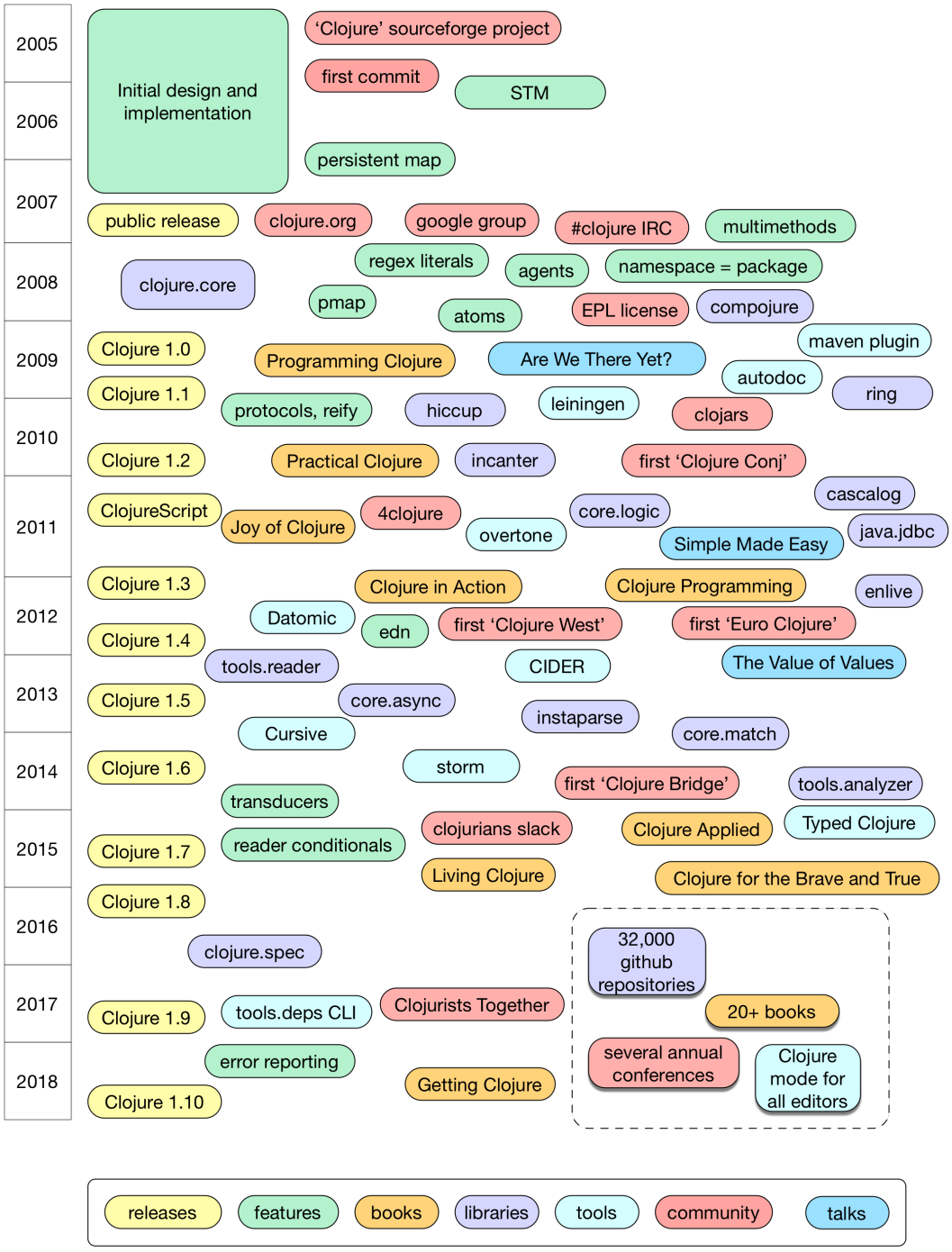


Fig. 1. Clojure timeline

Table 1. Clojure's data format ('edn')

Fixed and arbitrary precision integers	`1234, 12345678987654N`
Doubles	`1.234`
BigDecimals	`1.234M`
Ratios	`22/7`
Strings	`"fred"`
Characters	`\a \b \c`
Symbols	`fred, a.namespace/ethel`
Keywords	`:fred, :a.namespace/ethel`
Booleans	`true, false`
Nothing	`nil`
Regex patterns	`#"a*b"`
Lists: singly linked, grow at front	`(1 2 3), (fred ethel lucy), (foo 1 2 3)`
Vectors: indexed access, grow at end	`[1 2 3], [fred ethel lucy]`
Maps: key/value associations	`{:a 1, :b 2, :c 3}, {1 "ethel" 2 "fred"}`
Sets	`#{:fred :ethel :lucy}`

Another benefit of code-is-data is that it behooves the data structure set to include several language necessities as data types, e.g., symbols and keywords (Clojure has both), which ordinarily are ephemeral during compilation and not available to application programmers. This gives Lisp programmers the tools, and a recipe, for implementing their own languages with similar characteristics.

3.1.2 'read/print'. The separation of the data read/print system from compilation/evaluation has tremendous utility beyond its use for code. It serves as a human readable, text-stream-compatible serialization format. This makes the language data structure literals a first (and often winning) candidate for configuration files, program data storage formats and interprocess wire protocols. It was an objective of Clojure to take these use cases even further by adding more data structures, i.e. vectors and associative maps, to the core reader syntax. Clojure does occasionally use vectors in its language syntax, but that is not why they are there.

Data structure literals are a big win. There is a significant difference in the comprehensibility of a data literal vs the set of language instructions that might construct that same data. Furthermore, instructions are parochial to language functions whereas data structure literals are not. Data structure literals allow different programming languages to meet at a place of commonality (e.g., almost every language has numbers and strings, and sequential and associative data structures) for the purposes of interoperation, JSON [IETF 2017] being a prime example.

Table 1 enumerates the core data format of Clojure. Commas are optional and are considered whitespace, collections are heterogeneous, everything nests. Note that the reader can produce those data structures given that text *without invoking the compiler/interpreter*. Interpreting those data structures as the Clojure language is an entirely orthogonal process.

This data format of Clojure was later dubbed 'edn', extensible data notation, when tagged literals were added in Clojure 1.4 (see section 4.5). edn has reader/writer implementations in many of the major programming languages [edn 2019].

To the extent that the 'language' part of programming language design is about what people read and type, Clojure's design is captured in the design of edn. Programming in Clojure was going to be primarily about programming with these data structures not just in code, but pervasively.

3.1.3 Small Core Language. Lisps can be small; lambda, lexical bindings, conditionals, lists, symbols, recursion and voila, eval! While Clojure doesn't have a meta-circular definition, the language has a small core of special forms: `def`, `if`, `do` ,`let`, `fn` (lambda), `quote`, `var`, `loop`, `recur`, `try`, `throw` and `monitor-enter/exit`. To support host interop, there are '`.`' and `new` special forms as well as special consideration of fully-qualified symbols and symbols containing '`.`'. The 1.2 release of the language (2010, section 4.3) included facilities for defining data structures (`deftype/defrecord`), polymorphic functional abstraction (`defprotocol`) and the *implementation* of abstractions (both protocols and host interfaces) via `reify`.

It was a design objective of Clojure that there not be much cognitive load from the language itself. I think Lisp does that well. A further benefit for language evolution is that most of the action is in libraries, not language features [Steele 1999]. This allowed the core to remain stable, and I have been very conservative about enhancements at the language level.

3.1.4 Tangible at Runtime. Like Common Lisp packages, Clojure's namespaces are reified collections — accessible at runtime and enumerable. This yields important benefits for introspection and instrumentation. It is possible to resolve symbolic names to vars and classes, find documentation, source and other metadata, and directly manipulate the symbolic naming system in ways that in other languages might be relegated to the compiler or loader. Clojure symbols, vars and namespaces differ substantially from CL packages though, in ways discussed later (in section 3.2.1).

3.1.5 REPL. It is common to conflate any interactive language prompt with a REPL, but I think it is an important aspect of Lisp REPLs that they are a composition of read-eval-print. From a language perspective, one aspect of supporting REPL-driven development is that there are no language semantics in Clojure associated with files or modules. While it is possible to compile and load files, the effect of such loading is always as if executing each contained expression sequentially.

3.1.6 Flexible. A final important characteristic of Lisps is that they omit a baked-in higher-level programming model, be that object-orientation, category theory, a logic system etc, and any static checking system built thereupon. This might originally have been inevitable given Lisp's primitive genesis, but continuing this policy for Clojure was a deliberate design decision and objective. I was and remain skeptical of the suitability to task of these models as regards the kinds of programs I have seen and worked on, and of the cost/benefit ratio vis-a-vis expressivity, coupling, extensibility and specificity. If you want one of these models for a project or part of a project they are all available in Clojure via libraries. It was a design objective of Clojure to both stay out of your way and give you enough by default that a larger built-in model was not necessary for getting common programming tasks done.

The tradeoffs of this flexibility are an increased reliance on documentation and testing, and fewer opportunities for optimization.

3.2 Lisp Choices

Clojure never intended compatibility with a predecessor Lisp like Scheme or Common Lisp, which left me with the task of deciding which way to go on several core Lisp decision points.

3.2.1 Lisp-1 and Macros. Clojure uses a single resolution mechanism for symbols in the function position and elsewhere, and is thus a Lisp-1 [Gabriel and Pitman 1988]. This is strictly less complex than Lisp-2, more conducive to programming with higher-order functions, less context dependent etc. The only exceptions to this are some of the symbols designating host interop accepted only in the function position, for which there are no meaningful corresponding values.

However, Clojure supports programmatic macros (arbitrary functions of data → data, such data including symbols) in the Common Lisp style, which has been considered to be problematic

without the separate function space of Lisp-2. I concluded that in Common Lisp the conflation of symbols and storage locations, and the interning reader, were the sources of the conflict. The interning reader is particularly problematic; it creates a stateful interaction between reading and the runtime environment, breaking the otherwise functional nature of code-text → data-structures. By manipulating the environment, the interning reader restricts and interferes with the macro system's ability to control evaluation.

In Clojure, symbols are a simple immutable type akin to strings, with no storage. While symbols have two parts, an optional 'namespace' part and a 'name' part, they are not inherently interned in packages/namespaces. During `read` there is no interning nor resolution of symbols. The resolution of symbols to `vars` (Clojure's separate storage location type) generally happens during compilation.

A second place where resolution of symbols occurs is in Clojure's version of *quasiquote*, called *syntax-quote* and designated by '`'. Syntax-quote will take a form containing symbols and replace any unqualified symbol (i.e. one without a namespace part) with a fully-qualified one, where the namespace portion corresponds to the mapping in the namespace where syntax-quote was called. Thus a macro's use of the symbol `rest` (a Clojure core function) within a syntax-quote will be replaced with `clojure.core/rest` and will be free of conflict when that macro is expanded in a context with a different or local binding for `rest`. If one wants to introduce conflict-free local names Clojure has `gensym`. Taken together, this has worked out to be a quite satisfactory recipe for avoiding the so-called "hygiene problem" [Kohlbecker et al. 1986]

3.2.2 Reader Macros. One thing conspicuously missing from Clojure are Common Lisp's reader macros. While they fully empower the user to create a language of their own specification, such languages are islands. Critically, they require the presence of specific code to read particular data, such requirement being in direct conflict with some of the benefits of independent read/print enumerated above. I saw reader macros in the CL style as being in direct conflict with library development, interoperability, Clojure data (`edn`) as a wire format etc. I am certain that had Clojure had reader macros, and users availed themselves of them, the large, composable, data-driven library ecosystem that arose around Clojure would have been compromised or thwarted.

I am not opposed to extensibility, and added it to Clojure's data format with its christening as `edn` (the e stands for extensible) in the 1.4 release in 2012, described in section 4.5.

3.2.3 Tail Calls and Recursion. I very much would have liked to copy Scheme's 'proper tail recursion' [Steele Jr and Sussman 1978] and the elegant programming style it supports, but the JVM does not allow for that stack management approach in any practical way. On the other hand, I consider partial tail call optimization, e.g., of self-calls but not mutually recursive calls, to be a semantic non-feature. In addition, even with proper tail recursion, I thought it may be a point of confusion among users when exactly a call is in tail position and subject to the optimization.

Thus, while supporting recursive and mutually recursive function calls (without Scheme's unbounded promise), Clojure offers a limited but explicit looping recursion construct called `recur` and an accompanying binding/scope construct called `loop`. `recur` takes arguments matching the bindings in the enclosing `loop`, or the function head if no enclosing `loop`. Appendix A has more details on Clojure syntax:

```
(defn factorial [n]
  (loop [cnt n, acc 1]
    (if (zero? cnt)
      acc
      (recur (dec cnt) (* acc cnt)))))
```

The `defn` form above defines the function `factorial` in the namespace in which the `defn` form appears (e.g., 'user'). `factorial` takes a single argument 'n'. The `loop` special form takes one or more initial name to value bindings, in this case binding 'cnt' to the value of 'n' and 'acc' to 1. `if` takes a *test* expression, a *then* expression and an *else* expression. The *then* case here terminates the recursion, returning the value of the accumulator 'acc' from the function. The *else* expression is a `recur` that will return control to `loop`, with fresh bindings of 'cnt' to `(dec cnt)` and 'acc' to (* acc cnt). One could then invoke `factorial` in the namespace in which it was defined as follows:

```
;; lines beginning with user=>, other=> or => represent entries at the REPL
;; and are followed immediately by the REPL response,
;; a printed representation of the evaluation result
user=> (factorial 4)
24
```

or from another namespace with qualification:

```
(in-ns 'other)

other=> (user/factorial 4)
24
```

An advantage of `recur` is that the compiler reports an error if it occurs in other than a tail position. This has been helpful in practice—imperative programmers struggling to find how to do iteration search for 'loop' and find it, then get a gentle introduction to its recursive/ functional nature and help getting it right (the tail check).

3.3 Core Differences

Setting aside being hosted, the above differences with other Lisps were, I thought, insufficient to justify making another Lisp dialect from scratch vs building one atop Scheme or Common Lisp. There are several key areas where Clojure differs at its core from Scheme and Common Lisp and thereby justifies its existence.

3.3.1 Abstractions at the Bottom. Due to their history and heritage, Lisps prior to Clojure, and their standard libraries, have been built on concretions: primarily the mutable `cons` cell and lists constructed thereupon. While one can't fault the choice during the early days of Lisp, we now have sufficient technology (high performance runtime polymorphic dispatch, thanks OO!) to support different choices about what should be at the bottom. One positive aspect of years of OO programming is that it left me with a strong aversion to defining libraries in terms of concretions. I simply couldn't imagine choosing a `cons` cell with mutable `car` and `cdr` as the basis for a language or library in 2005, and so I did not.

It was my objective to tweak Perlis [1982] "It is better to have 100 functions operate on one data structure than to have 10 functions operate on 10 data structures." with "It is better to have 100 functions operate on one data abstraction than to have 10 functions operate on 10 data structures." For Clojure I built several such abstractions underlying its standard library covering sequentiality, collections, keyed access, etc. The abstraction replacing the cons cell is *seq*.

3.3.2 Seqs and Laziness. The idea behind seqs is to replace the concrete singly-linked list with an abstraction over all collections whereby they provide sequential access to their contents.

Given some collection-like thing `c`, `(seq c)` returns a *seq object* positioned 'at' the first element in a sequence of its elements, or `nil` if the collection has no elements.

Given some seq object `s`, `(first s)` returns the value at that point in the sequence, and `(rest s)` returns something on which you can call `seq` representing the rest of the sequence. Thus `first` and `rest` are an abstraction of the *interface* of cons cells.

Of course, a collection could satisfy the `seq` interface by immediately constructing a singly-linked list of its contents, but seqs had to satisfy a second important use case. They had to replace iteration. Not all collections reside in memory, and imperative programmers are used to using (mutable, stateful) iteration for accessing collection-like contents a step-at-a-time without forcing them to exist all at once in memory.

Clojure is strict, so I decided to provide a lazy implementation of the seq interface to support these use cases, and to encourage its use pervasively so that all of the standard sequence library, and everything built upon it, would be capable of manipulating collections larger than memory, infinite sequences etc. This was just library code, no language additions were necessary. The lazy seq object utilizes the expected memoizing thunk, and I provided a `lazy-seq` macro with which one could wrap ordinary recursive code. Here `interleave-2` takes two collections and alternates their elements until either is exhausted:

```
(defn interleave-2
  "Returns a lazy seq of the first item in each coll, then the second etc."
  [c1 c2]
  (lazy-seq
   (let [s1 (seq c1)
         s2 (seq c2)]
     (when (and s1 s2)
       (cons (first s1)
             (cons (first s2)
                   (interleave-2 (rest s1) (rest s2))))))))

=> (interleave-2 [:a :b :c] [1 2 3])
(:a 1 :b 2 :c 3)

;; when passed two infinite sequences, interleave-2 returns an infinite sequence
=> (take 10 (interleave-2 (cycle [:a :b :c]) (range)))
(:a 0 :b 1 :c 2 :a 3 :b 4)
```

The body of `lazy-seq` is not evaluated until `seq`/`first`/`rest` is called on object it produces. Thus `interleave-2` can consume and produce infinite collections, e.g., above where both `cycle` and `(range)` return infinite sequences.

Here is an example of `lazy-seq` use with some host interop, lines from files, which may not fit in memory:

```
(defn line-seq
  "Returns the lines of text from rdr as a lazy sequence of strings."
  [^java.io.BufferedReader rdr]
  (when-let [line (.readLine rdr)]
    (cons line (lazy-seq (line-seq rdr)))))
```

`^java.io.BufferedReader` is a metadata type hint on the single parameter 'rdr'. Note here that a BufferedReader is a stateful Java object and `.readLine` mutates it. Because lazy sequences memoize their evaluations, the reader is consumed only once, no matter how often the resulting seq is consumed. While the programmer must manage the lifetime of, and avoid sharing, the underlying

stateful source, it is nevertheless quite useful to provide such bridges between the stateful world of Java and immutable, lazy sequences and the large core library that can consume them.

In 2009 I enhanced the compiler to aggressively null-out stack references to no-longer-used variables, thus ensuring the JVM understood they were available for incremental garbage collection, avoiding any memory explosion related to laziness. Fortunately it was never a promise or objective of this system that it be 'strictly' lazy, i.e. that it would calculate exactly one 'next' item and only upon need. That weak promise enabled a later optimization that allowed for 'chunked' sequences that calculated up to 32 items at a time, substantially amortizing the overhead of thunking.

You might notice `cons` above, and yes, Clojure has (immutable) `cons` objects, but they implement the seq interface like everything else and enjoy no privileged role. By pervasively supporting the seq abstraction on Clojure's and host collections, the dozens of Lisp/fp functions ordinarily associated with lists (map, reduce, filter et al) work instead on everything.

Clojure does have an abstraction of "adding to a collection", called `conj` (conjoin), which works on all the collections (lists, vectors, maps, sets).

3.3.3 Functional. Programming in a functional style is possible to some extent in many languages, but what matters is what a language makes idiomatic. It was a core objective of Clojure to simplify programming, and that is primarily reflected in a desire to support and ensure that the bulk of any Clojure program was a set of pure functions taking and returning immutable values.

In Scheme and Common Lisp, a functional style is supported and used, especially when manipulating lists. But it is just a convention; the data structures are mutable. The functions supporting arrays and hash tables are imperative. In Common Lisp e.g., it is easy to construct a largely-functional prototype using the list functions like `assoc`. But the prototype can't scale to larger data sets because of the poor performance of lists when used as associative data structures. At that point you have to move to hash tables which (a) require different code and (b) are mutable. Thus I do not think these Lisps support or encourage functional programming in the large.

In Clojure, function arguments and let bindings are immutable. All of the core data structures: lists, vectors, maps and sets, strings, and all the scalar types, are immutable. The vast majority of the standard library are pure functions. Clojure not only supports functional programming, it ensures that functional programming is idiomatic and scalable, i.e. as a general rule functions are only supported on data structures for which they are efficient—no `assoc` on lists! While Clojure is not pure, it is arguably a functional programming language where prior Lisps are not.

That said, there are also levels of functional programming. Functional programs are desirable because they are simpler and easier to reason about, due to being mathematical i.e. they are free of time and place. This is the benefit Clojure pursues. There are those for whom functional programming is also about reasoning about programs mathematically i.e. via proof. Clojure is not concerned with proof in programming.

3.3.4 Summary. When you take the broadest notion of Lisp, programming in and with fundamental data structures, Clojure is both clearly a Lisp and seeking to extend the Lisp idea. In being built on abstractions and strongly focusing on functional programming, it is a novel Lisp.

3.4 Clojure's Data Structures

While linked lists are available and can be implemented in most programming languages, and have a certain elegance and functional heritage, their use is dominated in professional practice by the use of vectors/arrays and hashtables/associative maps. Of course, lists are sequential like arrays and one can access the nth element, and e.g., Common Lisp has a notion of association lists and a function 'assoc' for creating mappings and thus treating a list of pairs like a map. But it highlights an important aspect of programming and difference from mathematics in that merely getting the

right answer is not good enough. Not all isomorphisms are viable alternatives; the leverage provided by indexing support matters. I felt Clojure would be a non-starter for practitioners without credible substitutes for O(1) arrays and maps.

3.4.1 Persistence and Immutability. What I thought would be a simple matter of shopping for best-of-breed functional data structures ended up being a search and engineering exercise that dominated my early Clojure work, as evidenced by the gap in commits during the winter of 2006/7 (figure 2). I started by looking at Okasaki [1999], and found the data structures too slow for my use, and felt some of the amortized complexity would be difficult to explain to working programmers. I also concluded that it mattered not at all to me that the implementation of the data structures be purely functional, only that the interface they presented to consumers was immutable and persistent.

This led me to look at the fat node approach to making data structures persistent [Driscoll et al. 1989]. However, the in-place updating required to obtain the advertised bounds made them impractical for multithreaded use, where extensive synchronization would be required. So I *did* in fact care that, *post construction*, the structures would never be subject to mutation—there could be no faking it.

I then set out to find a treelike implementation for hash maps which would be amenable to the path-copying with structural sharing approach for persistence. I found what I wanted in hash array mapped tries (HAMTs) [Bagwell 2001]. I built (in Java) a persistent implementation of HAMTs with branching factor of 32, using Java's fast `System.arrayCopy` during path copying. The node arrays are freshly allocated and imperatively manipulated during node construction, and never mutated afterwards. Thus the implementation is not purely functional but the resulting data structures are immutable after construction. I designed and built persistent vectors on similar 32-way branching trees, with the path copying strategy. Performance was excellent, more akin to O(1) than the theoretical bounds of O(logN).

This was the breakthrough moment for Clojure. Only after this did I feel like Clojure could be practical, and I moved forward with enthusiasm to release it later that year (2007).

Clojure's success with and evangelism of persistent HAMTs influenced their subsequent adoption by Scala (release 2.8, in 2010), Haskell (unordered containers, in 2011), Erlang (large maps, release 18, in 2015) and others. Bagwell and Rompf [2011] went on to refine Clojure's approach to immutable vectors with RRB-Trees.

3.4.2 Information Programming—"just use maps". A key question for language choice (and thus software design) is: how well do the primary language constructs map to your primary problems? If your program deals with information, these are among your primary problems: information is sparse, incrementally accumulated, open/ extensible, conditionally available, formed into arbitrary sets in different contexts, merged with other information etc. Thus the answers to these questions become important: Can you determine what information is present? Is there an algebra for information selection? Merging? How difficult is it to accumulate information in a processing pipeline? Are attributes first-class? Is there a system for avoiding attribute name conflicts?

In my experience, statically typed class/record systems are a mismatch for these information management tasks. Aggregate types (classes/records) are the primary drivers of attribute semantics rather than merely being context-dependent aggregations. They are fully enumerated and closed. Though optionality/availability of a field/attribute is not a property of a data structure (it's a property of a *context of use*), you must declare things as Nullable or Maybe once and for all. Attribute names are parochial. Attribute/field names are often not first class. There is no algebra for merging/selection. Incremental information acquisition is hard. Mapping tools (like ORMs) to/from the parochial type models are common and necessary. Type errors, pattern matching errors and

refactoring tools are venerated for facilitating change instead of being recognized as underscoring (and perhaps fostering) the coupling and brittleness in a system.

I should note that the same problems often arise with traditional SQL database systems, where tables are the fully-enumerated culprits, and column and table names are not first class, nullability is a fixed decision, sparseness and openness are problematic, etc.

In information systems large and small, one is often forced to abandon the class/record system and just start using maps, which often have no syntax or other language support. And on the database side, one often starts using generic entity/attribute/value (triple) tables to get the flexible information support one needs. By the time I started Clojure I was pretty tired of all this mismatch with common needs, and the programmer effort wasted presuming that language models were right and all this marshaling and specificity was necessary or productive.

At the same time, I had been doing some research into RDF [W3C 2014] , a system designed around the needs of open, extensible information management. There, the semantics of properties (attributes) are inherent in the attributes themselves, not determined by their presence in some named, fully enumerated aggregate/class/record type. Aggregation is ad hoc. Properties are named by URIs and thus conflict-free. The RDF design resonated with me and was a big influence on both Clojure and Datomic (section 4.6).

I spent some time experimenting with various immutable record ideas but never ended up with something that added significant value to just using maps. Thus I decided I wanted programming with maps (and vectors and sets) in Clojure to be as first-class and native feeling as programming with lists had always been in Lisp.

A key question was: one map or two? There are two very distinct use cases for maps: potentially large, homogeneous, collection-like maps, e.g., mapping everyone's username to their site visit count, or comparatively small heterogeneous information-maps, e.g., a bunch of things we know about a particular user in a particular context, like the email address, password, userid etc. I wanted only a single map literal syntax and library for maps. As with seqs, the maps were built behind several abstractions. Thus, maps can start with a representation optimized for small maps and switch to a representation optimized for large collections if and when they 'grow' beyond a certain size.

In addition to the map read/print support, Clojure has a large functional standard library for dealing with maps, with functions for creating maps (sorted and hashed), associating and dissociating keys with values, merging, lookup, key selection and enumeration, nested map selection and update, sequencing (a seq of a map yields a sequence of key-value tuples), as well as a library implementing relational algebra on sets of maps. Of course, any map library could have these things, but no language at the time had them as pure functions of high-performance persistent HAMTs. In addition, a large subset of the map functions also work on vectors, which are, of course, also associative.

There were a few language features that further aided the 'native programming with maps' feel. In Clojure, invoke-ability is an abstraction, not limited to functions. Thus the associative persistent collections (vectors and maps) can be and are invokable; they are mathematical functions of their indexes/keys respectively. That means maps and vectors can appear in the first (ordinarily, function) position of an expression and can be passed as if a higher-order-function anywhere functions are accepted (map, filter, etc).

This abstraction of invoke-ability is just another example of detaching prime facilities from specific representations (e.g., when only 'lambda objects' can be invoked). In practice it leads to more succinct and direct code, fewer wrappers (and the allocation and indirection overhead thereof), and fewer awkward moments where, having been handed a map wrapped in an 'accessor' lambda, you want access to the wrapped/closed-over (and thus encapsulated) value. It is also useful to be

able to define invoke-able things that satisfy interfaces and protocols that ordinary functions do not:

```
;; visit-counts is a map
=> (def visit-counts {"ethel" 11 "lucy" 26 "ricky" 5})

;; map used in function position
=> (visit-counts "ethel")
11

;; map passed as function to higher-order function
=> (map visit-counts ["ricky" "lucy"])
(5 26)
```

Keywords are also an integral part of the 'just use maps' story. Keywords are a symbolic scalar type, distinct from symbols, with a high-performance (identity) equality check. As distinct from symbols, which are resolved to bindings during evaluation, keywords always evaluate to themselves, and thus don't require quoting. Thus keywords are the preferred first-class attribute names: keys in information maps. Keywords (and symbols) also implement the invokable abstraction. They are both functions of maps, and return the value 'at' themselves in the map, or nil if not present:

```
;; user is a map
(def user {:email "fred@example.com"
           :userid "freddy"
           :phone "555-1212"})

;; basic map enumeration - the keys
=> (keys user)
(:email :userid :phone)

;; and corresponding values
=> (vals user)
("fred@example.com" "freddy" "555-1212")

;; seq of a map is sequence of key/value pairs
=> (seq user)
([:email "fred@example.com"] [:userid "freddy"] [:phone "555-1212"])

;; get a projection
=> (select-keys user [:email :userid])
{:email "fred@example.com", :userid "freddy"}

;; merge two maps to make a third
=> (merge user {:prefs/favorite-food "pizza"})
{:email "fred@example.com", :userid "freddy",
 :phone "555-1212", :prefs/favorite-food "pizza"}

;; keyword used as function (of a map)
=> (:email user)
"fred@example.com"
```

```
;; this key is not present, not an error
=> (:prefs/favorite-food user)
nil

;; keyword passed as function to higher-order function
=> (map :userid [user {:email "ethel@example.com" :userid "ethelm"}])
("freddy" "ethelm")
```

Note that while most of the keywords above are not namespace-qualified, and thus have context-dependent semantics, Clojure *does* support multi-segment namespace qualifiers and aliases thereof for both symbols and keywords (e.g., `:prefs/favorite-food` above). When combined with Java package name reverse-domain conventions (e.g., `:org.my-org/my-name`, as is encouraged in Clojure) keywords are as useful for defining openly extensible, mergeable, conflict-free keysets as are URIs for RDF properties.

It is also important to note the difference between invokable keyword data values and using actual functions as field accessors, as is the case in some record systems. Being data, a keyword can be read from a file or be directly entered by a user or travel over wires without compilation or reference to any codebase.

These facilities make Clojure a good language for information programming. Specifically, they make it easier to write context-independent, generic, loosely coupled information processing programs that can be driven by data. Keys and maps are first-class values not encoded in specific program types, readable from text without specific program code, and can be produced by different languages. This is the way that large, loosely coupled programs that interoperate over wires are constructed, and in my opinion it's the way that large loosely coupled programs that don't (yet) operate over wires should be constructed as well.

3.4.3 Metadata. Another thorny problem in information systems is the conveyance of the *context* of information; provenance, timing, verification status, location, etc. This becomes harder still as we start treating everything as first-class values with the expected equality semantics. If we put context information inside the values then it impacts equality, which is not generally what we want. Such context information is something you often don't want to *see* either.

My solution for Clojure was to have a first-class notion of *metadata*—information about information. Clojure's collections and symbols all support an associated metadata map, with a functional API for setting/updating and reading the metadata. Critically, metadata does not impact equality semantics and does not ordinarily print unless requested:

```
=> (def mvec (with-meta [42] {:from "fred"}))
#'user/mvec

;; metadata doesn't ordinarily print
=> mvec
[42]

;; 'meta' obtains it
=> (meta mvec)
{:from "fred"}

;; metadata does not impact equality
=> (= [42] mvec)
true
```

There is also reader support for adorning data with metadata at read time, without code or evaluation. The caret causes the reader to read the next map/symbol/keyword and use it to construct the metadata for the next object read, as follows:

```
;; a map following the caret is merged
=> (meta ^{:from "fred"} [42])
{:from "fred"}

;; a symbol becomes the value of :tag
user=> (meta (quote ^String s))
{:tag String}

;; a keyword becomes a key with value of true
=> (meta ^:verified [42])
{:verified true}
```

Clojure uses metadata in several ways. When reading files, the reader will attach position (line and column) information to the read forms as metadata. Metadata on vars captures documentation and other interesting information:

```
=> (defn foo "does nothing" [x])
#'user/foo

=> (pprint (meta (var foo)))
{:arglists ([x]),
 :doc "does nothing",
 :line 50,
 :column 1,
 :file "NO_SOURCE_PATH",
 :name foo,
 :ns #object[clojure.lang.Namespace 0x1252b961 "user"]}
```

Clojure supports metadata for type hints which can be used to generate faster code:

```
;; len is a function that expects a String
(defn len [x]
  (.length x))

;; len2 says so via type hint metadata
(defn len2 [^String x]
  (.length x))
```

The type hint on the symbol `x` in `len2` is used by the compiler to issue a non-reflective call to `.length` and an efficient runtime-checked cast of the argument, making len2 10x faster. One nice thing about the metadata strategy for type hinting is that metadata flows through macro transformations transparently—macros don't need to understand or propagate type declarations, and no additional structure or nesting is introduced.

The metadata support is a general facility and is open to use by applications for any context-management scenarios.

3.4.4 Equality. Clojure supports for its own types the expected equality semantics akin to Baker's "EGAL" [Baker 1993] (identity semantics for mutable things, value semantics for immutable ones), modulo metadata as discussed. Collections introduce another important decision point: do distinct collection types create distinct equality partitions? Either you are programming with values or you are programming with types and Clojure very much resides in the values camp. Type differences are often implementation or performance details, and should not corrupt value equality.

In Clojure I eventually decided to support the categoric partitioning of collection types as per the design of Java's collection library, with which I agree: collections are either *sequential*, *sets* or *maps*, equality is supported within but not between partitions, and anything more specific about the collection types is irrelevant for equality. Thus lists, seqs and vectors with the same elements in the same order are equal, but a set and vector with the same elements are not:

```
;; lists, vectors and lazy-seqs are all sequential
=>  (= '(1 2 3) [1 2 3] (map inc (range 3)))
true

;; sets and vectors are in different equality partitions
=> (= #{1 2 3} [1 2 3])
false
```

Similarly for numbers the precise integral types (fixed and variable precision integers and ratios) support value equality, but floating point numbers form a different partition:

```
;; integer, ratio, biginteger
=> (= 42 84/2 42N)
true

;; integer and floating point are different partitions
=> (= 42 42.0)
false
```

Unfortunately Java's boxed numeric types have type-specific equality, and Clojure needed to 'fix' equality there:

```
;; .equals is Java equality
=> (.equals (Integer. 42) (Long. 42))
false

;; Clojure equality
=> (= (Integer. 42) (Long. 42))
true
```

While '=' is overwhelmingly primary, it is still not the only comparator. '`.equals`' is supported for host compatibility (with host-dictated semantics), '==' is available for even more permissive numeric comparisons and '`identical?`' is available and useful for optimization. That said, Clojure's rigorous yet polymorphic equality semantics mean that even for deeply nested heterogeneous data structures '=' "just works", an essential foundation for value-oriented functional programming.

3.4.5 Summary. Clojure's data structure support makes it possible to directly take on a problem using primarily the provided data structures and library. Such code is succinct (due to data literals), performant (persistent HAMTs have best-in-class performance for persistent maps), functional (due to immutability) and idiomatic (data literal support begets use of the libraries and the functional strategy).

3.5 Clojure and the Host

Libraries are power. If a programmer needs a utility for which they have no library, then they lack the power to get their primary work done. A new language has not had the time nor user base to generate many libraries. Thus adopters of new programming languages lack power, often forever. It was a primary objective of Clojure to eliminate that problem. I wanted to use it, for real, for work, right away, and convince others to do the same. No professional programmer can afford to be off the grid generating power by burning trees they grew themselves, i.e. writing everything from scratch.

Large programmer ecosystems are like power plants. Long lived, popular languages with many users and vibrant open source communities have lots of libraries. The most popular of those libraries in turn have many users, an indicator but not guarantor of their quality and practicality. Thus it is quite common for small/new languages to have some sort of foreign function interface (FFI) over which they can create a bridge to enable parasitic consumption of the larger community's libraries. FFIs can also be used as a bridge between higher level but less efficient languages and subroutines written in more efficient ones.

Often the bridge is a long one, e.g., when bridging between a language with garbage collection and one with manual memory management, or into a language with an elaborate type system wherein foreign things need extensive description in order to participate in type checking.

The idea for Clojure behind being *hosted* was that there would be no overhead, conversion, friction, wrapping, declarations et al going to and from the larger ecosystem because there would be no *to* and *from*. A Clojure program would be native to that ecosystem, tissue rather than parasite. Thus Clojure and programs written in Clojure *are* Java libraries and bytecode when on the JVM, *are* .Net assemblies on the CLR, and *are* Javascript on Javascript engines.

This idea extends beyond merely compiling to the native bytecode/language, because it is quite possible to do that and end up with something that looks to the host quite foreign (like cancer). This can preclude bidirectional integration—not only the host's ability to call the hosted language but also language's ability to consume host functions that have requirements (e.g., of arguments) that the language's values can't satisfy.

3.5.1 Presumptions of the Host. Clojure initially targeted the JVM [Lindholm and Yellin 1999] and the CLR. The first Clojure compiler was a Common Lisp program that generated Java and C# code. Java and C# are extremely similar and their intersection determined the first cut of what Clojure expected, but did not necessarily require, of a host.

Clojure *needs* the host to be garbage collected and have some high performance mechanism and/or optimization for runtime polymorphic dispatch. It *expects* the host to be object-oriented insofar as there are objects with methods or fields, and might have constructors of some sort. It expects, but certainly does not require, that the host might have named classes, types and interfaces, and use inheritance to implement runtime polymorphism.

This might sound extremely imprecise and open, and it is, by design. Precision's first cut is flexibility. The design was tested when in 2011 I started work on ClojureScript (see section 4.4), an implementation of Clojure on Javascript hosts. Javascript has a substantially different object model than do Java and C#, but the interop support mapped where needed very well, requiring only a single, backwards compatible extension to assist in disambiguating fields and method calls, since 'methods' in JS are simply fields containing functions.

Though I started by targeting both the JVM and CLR, via generation of Java and C# source, by the end of 2006 I had decided I wanted to accomplish twice as much rather than do everything twice, so I decided to target only the JVM moving forward. That choice was driven by the much stronger open source library ecosystem of Java and the more dynamic nature of the JVM, e.g.,

HotSpot optimizations [Click and Rose 2002]. The compiler was moved to Java and eventually (prior to release) generated bytecode directly. Only interop on the JVM is discussed below.

3.5.2 Consuming Host Libraries. Clojure has syntax for creating new instances, accessing members (methods and fields, instance and static), referring to classes, imperative assignment to fields, and access to arrays. The host support remains a prefix system (i.e. methods are the 'operators' and come first, before the target object, which is treated as an implicit first argument). There was no attempt to make first-class values for things which on the host were not (e.g., Java methods):

```
;; instance method invocation
=> (.toUpperCase "fred")
"FRED"

;; classnames yield Class values
=> (.getName String)
"java.lang.String"

;; constructor syntax 'Classname.'
(def p (java.awt.Point. 1 2))

;; field accessor
=> (.-x p)
1

;; static method
=> (System/getProperty "java.vm.version")
"25.151-b12"

;; static field
=> Math/PI
3.141592653589793
```

Additional macro support yields host interop programs that are more succinct than their native language equivalents, including requiring fewer parentheses (Lisp's revenge):

```
// Java - so many parentheses!
a.foo().bar().baz();

;; Clojure calling Java
(-> a .foo .bar .baz)

// Java - statements
X a;
a.init();
a.setThis(42);
a.setThat(21);
return a;

;; Clojure calling Java - an expression
(doto a .init (.setThis 42) (.setThat 21)) ;; yields a
```

3.5.3 Sharing Host Types and Abstractions. To maximize interoperability and performance, wherever the semantics agree, Clojure's scalar types are host types. So, on the JVM, Clojure's strings are `java.lang.Strings`, the fixed precision numbers are Java `Doubles` and `Longs`, the booleans are `Booleans` and `nil` is Java `null`.

I was happy to find that Java's collection class library authors a) used interfaces, and b) declared all of the mutating methods of the collection interfaces *optional*. Thus I was able to implement these interfaces for Clojure's immutable collections, opting out of mutating methods. Clojure's collections implemented `java.util.Collection` early on, and came to support the more specific interfaces `java.util.List/Set/Map` as appropriate. That means Clojure users can be happily programming with Clojure's data structures and 'drop down' to Java calls, passing them intact:

```
;; Java method requiring Java List, passing Clojure vector
=> (java.util.Collections/binarySearch [2 4 7 9 43 76] 7)
2

=> (class [2 4 7 9 43 76])
clojure.lang.PersistentVector

=> (instance? java.util.List [2 4 7 9 43 76])
true
```

Similarly, Clojure's functions implement `Callable` and `Runnable` and Clojure's extended numeric types implement `java.lang.Number`.

3.5.4 Extending Clojure's Polymorphism to the Host. Until version 1.2 and the introduction of protocols (see section 4.3), Clojure did not have a polymorphism construct of its own. Since the core data structures were written in Java, the natural representations for the initial abstractions were Java interfaces: `clojure.lang.IPersistentCollection, IPersistentMap, IFn, ISeq` etc. However interfaces suffer from the expression problem, and require cooperation of the class authors in deriving from any interface abstraction they intend to support. The system is closed. It was critical to the ideas behind Clojure that the core library be as polymorphic as possible, including reaching host constructs I could not change.

What mattered most was that the library be polymorphic, not what it took to make it so. So, under the hood of several of the library functions, especially the gateway functions like `seq`, I had to support many of the host types explicitly (as often as possible through interfaces, like `Iterable`, that were commonly supported). The net result is quite powerful; `seq`, and thus Clojure's entire sequence library, appears to work with everything—Clojure's seqs and collections, strings, Java arrays, all Java collections, Java maps, anything Iterable, etc. This maximizes the utility of the standard library, and increases the generality and applicability of code written in terms of it.

3.5.5 Extending Host Abstractions. Many host APIs require derivation from specific interfaces (and, distressingly, sometimes concrete classes) in order to interoperate. Because Clojure came without facilities for defining types and their bases, Clojure included `proxy`, and later in 1.2 `reify`, that let you generate an anonymous implementation of a Java interface for use with these APIs, each 'method' of which looks and behaves like a function.

3.5.6 Error Handling and Exceptions. Probably the biggest host construct that directly dictated to Clojure was exceptions. I experimented a bit with emulating Common Lisp's conditions system, but I was swimming upstream. The anticipated hosts all implement error handling and stack unwinding via try/throw/catch style exceptions. It is optimized, has tooling and security support, and is adequate. I just adopted it wholesale in Clojure.

3.5.7 Portability. Given the likely intimacy of any particular application with the host, it was never my intention that entire Clojure applications be portable between hosts. My objective was to support portability of the language, its libraries and programmer skills; *"I can use Clojure here"*. It is also important to recognize the portability of the hosts themselves, Java and Javascript, for example, have been extensively ported to different architectures and operating systems.

That said, portable libraries can be written on Clojure's core, non-trivial examples being `core.logic`, a Prolog-like relational, constraint logic, and nominal logic programming library; `test.check`, a property-based testing tool inspired by QuickCheck; Clara rules, a RETE-driven rules engine; and the ClojureScript compiler itself.

3.5.8 Runtime Model. A further benefit of being hosted is the reduced cognitive load of not introducing a new runtime model. Programmers need intuition about a language's runtime model—how it uses memory, how GC works, how threads work, etc, in order to be effective. And it takes years to develop that intuition around a model like Java's, which sits at the edge of tangibility. Higher-level programming language models often abstract away so much that it becomes impossible for a programmer to anticipate memory use or optimization. This is a barrier to practicality. Clojure's runtime model is the model of the host, and all hard-won intuition thereof applies.

3.5.9 Tools. Clojure had sophisticated tooling from early on. Once I started emitting the needed debug information (just after release, in 2007), all of the Java breakpoint/step debugging and profiling tools worked on Clojure programs. This is valuable tooling for a brand-new language to have.

3.5.10 Implementation Benefits. In addition to the features derived from being hosted, there were many implementation benefits. I didn't have to think at all about producing machine code, portability, garbage collection, optimization etc. Clojure is always compiled (on the JVM, to Java bytecode), there is no interpreter. From bytecode, the commonly used HotSpot runtime does an excellent job of runtime profiling, code generation, inlining and other optimizations, thus the Clojure compiler didn't need even basic optimizations, e.g., dead code elimination. It is a simple hand-rolled recursive descent compiler, written in Java.

The JVM's single-dispatch runtime polymorphism and optimization thereof is quite excellent, a prerequisite for Clojure situating abstractions at the bottom of its core libraries and data structures. The JVM runtime is also highly dynamic, which supports Clojure's dynamic nature; new classes can be defined and loaded on the fly, classloaders are tangible and extensible. And the JVM has low-cost and verified runtime type checking on downcasts. Thus while Clojure is dynamic, it's not untyped "get it right or segfault". All objects are invariably coerced into something specific before use, e.g., if you try to add string to a number you will get a runtime exception telling you your string is not a number.

The list goes on and on: bytecode verification, security models and sandboxing, sophisticated GC etc. The JVM is a production runtime representing dozens (hundreds?) of person-years effort that no new language is likely to replicate without significant corporate sponsorship (as had .Net and the CLR).

3.5.11 Summary. Clojure's approach to being hosted was a significant factor in its adoption, according to user experience reports and survey results. Early adopters could 'sneak in' Clojure as just another Java library. In adopting Clojure they got an immediate increase in power, as Clojure's functional library and data structures were added to their toolkit without any loss. And the benefits accrue on an ongoing basis; as Java API support is a top priority for new services, Clojure users can access these services as soon as Java can.

3.6 State

While I did believe, and it has been true in practice, that the vast majority of an application could be functional, I also recognized that almost all programs would need some state. Even though the host interop would provide access to (plenty of) mutable state constructs, I didn't want state management to be the province of interop; after all, a point of Clojure was to encourage people to stop doing mutable, stateful OO. In particular I wanted a state solution that was much simpler than the inherently complex locks and mutexes approaches of the hosts for concurrency-safe state. And I wanted something that took advantage of the fact that Clojure programmers would be programming primarily with efficiently persistent immutable data.

3.6.1 References. SML references [Milner et al. 1997] were the inspiration for Clojure's state management, essentially boxes that could point to (presumably immutable) values. I saw them as a fundamental *identity* construct and a great way to disentangle identity from value, as they are commonly conflated in OO languages. Unfortunately SML refs are simply pointers, manipulated via assignment, with no concurrency semantics and all the attendant read-modify-write race conditions.

The basic idea was to have a variety of references that were concurrency safe, never requiring user locking, with different update semantics varying in two dimensions:

Synchronization Will the update have happened when the update function returns, or will it happen sometime later?

Coordination Can the update of more than one ref be made atomic?

All the reference types would have unique update operations, because the semantics differed, but they all shared an abstraction for dereferencing, thus code that only *read* references would not care which kind they were. Dereferencing got some syntactic sugar in the reader whereby `@aref` is read as `(deref aref)`. Another important principle was that all references be available for (fast) reads at all times, i.e. outside of transactions. Update should not impede observation. While the update operations would be unique, they would all share the same signature:

```
(an-update-op aref fn-of-state-and-args-returning-newstate & args*)
```

with a shared guarantee that, if and when an update was applied, it would be atomic—no broken pointers or values.

It is also important to note that none of the concurrency support required anything of the core language, it is all library code.

3.6.2 'refs' and the STM. Obviously the coordinated case is the tricky one, and I only supported the synchronous flavor. Since the goal was to turn multiple updates into a single unit of work, there would need to be a scope for that work and the notion of transactions naturally arose. I looked into the software transactional memory (STM) research and found that most of the work involved trying to superimpose transactions over otherwise ordinary imperative mutable state code in existing programming languages, with not great results. More similar to my problem was Haskell's STM [Harris et al. 2005], but it had composability objectives I didn't share.

Most important, none of what I saw co-aligned with the way I thought about the problem, which was that this was not different from the problems faced by database transactions, with which I was very familiar. Database implementors have been tackling transactions, serializability, read semantics etc and the optimization thereof for a long time [Gray and Reuter 1992].

In reading the STM literature one problem I saw often was read-tracking, which I definitely wanted to avoid for performance reasons. In the database world, one technique for avoiding read tracking and read locks is multiversion concurrency control (MVCC) [Bernstein and Goodman 1983]. It seemed to me that MVCC was a great fit for persistent data structures which supported

efficient update with structural sharing, making versioning cheap. So I designed and built an STM around MVCC.

In basic MVCC only writes are tracked, and all reads are done with a consistent basis (as of the transaction start, and incorporating within-transaction writes). However MVCC can be subject to write-skew, where the basis of a write to X is a read of Y, and Y is not otherwise written, and thus not ensured consistent at transaction end. Clojure's STM offers an `ensure` operation for opt-in tracking of such reads when needed.

The fundamental update operation of refs is `alter`. As a further optimization, a second update operation `commute` is supported. `commute` can be used when the state transformation function is commutative, and allows two overlapping transactions that `commute` the same ref to both proceed (whereas with `alter` the transactions would be serialized).

Transaction scope is established with `dosync` and attempts to `alter/commute` refs outside of a transaction fail. Consistent (though uncoordinated) fast reads of refs can happen outside of transactions.

3.6.3 Agents and Atoms. In 2008 I added two more reference types: *agents*, which covered the asynchronous uncoordinated case, and *atoms*, which covered the synchronous uncoordinated case. Agents enqueue commutative operations for later application, while atoms encapsulate an atomic compare-and-swap loop. Atoms are the fastest managed state primitive in Clojure.

3.6.4 Vars. Clojure also has a global variable system: *vars*. These references are interned in namespaces and are the storage locations to which free symbolic references in code are resolved and bound. Vars are created and interned via `def` and its various flavors. Vars can be dynamically rebound on an opt-in basis. The primary purpose of the var system is for references to functions, and their variable nature is what supports dynamic development.

3.6.5 Summary. Taking on the design and implementation of an STM was a lot to add atop designing a programming language. In practice, the STM is rarely needed or used. It is quite common for Clojure programs to use only atoms for state, and even then only one or a handful of atoms in an entire program. But when a program needs coordinated state it really needs it, and without the STM I did not think Clojure would be fully practical.

Having a credible, efficient, functional state management story that demonstrated the many advantages of immutability vs mutable objects plus locking was critical to my early evangelism of Clojure. Being able to talk about identity, state, and value with these constructs helped me describe functional programming to OO developers as something simple, approachable and viable.

3.7 Polymorphism

It was my intent that Clojure have a predicative dispatch system [Ernst et al. 1998]. None of my experiments produced a system with good enough performance to live at the 'bottom' of other constructs. Instead I developed multimethods, inspired by Common Lisp's generic functions. A multimethod defines a single named function, with which one can associate an open set of implementation 'methods'. Each multimethod is defined with a dispatch function, a function of *all* of the arguments, and methods are associated with particular return values of the dispatch function. A multimethod can be assigned a default implementation which will be called when no there is no match:

```
;; Note: In this example, the keyword :Shape is being used as the
;; dispatch function, as keywords are functions of maps

;; area is a function defined by an open set of methods - a multimethod
;; dispatch is determined by the value returned by :Shape (a keyword used as a function)
(defmulti area :Shape)

;; the method that will be called when :Shape returns :Rect
(defmethod area :Rect [r]
  (* (:wd r) (:ht r)))

;; the method that will be called when :Shape returns :Circle
(defmethod area :Circle [c]
  (* Math/PI (:radius c) (:radius c)))

;; the method that will be called when no other dispatch applies
(defmethod area :default [x] :oops)

;; information vs objects
(def r {:Shape :Rect, :wd 4, :ht 13})
(def c {:Shape :Circle, :radius 12})

=> (area r)
52
=> (area c)
452.3893421169302
=> (area {})
:oops
```

Multimethods are quite flexible, and performance is good enough for many applications, e.g., multimethods are used for Clojure's polymorphic printing. But they cannot be made as fast as Java method dispatch. Thus the initial release of Clojure lacked a sufficient polymorphic dispatch system to define something like Clojure in Clojure.

4 EVOLUTION

The timeline in figure 1 shows the relative timing of releases, feature introductions, book publications, prominent libraries and tools, community forums and significant talks discussed below.

4.1 Fall 2007—May 2009, First Alpha through Clojure 1.0

I announced Clojure's availability on October 17, 2007, in the following message to the dozen or so members of the jFli mailing list:

> "As someone interested in Foil or jFli, I thought you might want to know about my latest project — Clojure, a dialect of Lisp for the JVM. It's currently alpha, but fairly complete. I'm looking for some feedback from some intrepid folks willing to kick the tires."

with links to a web site (clojure.org) and a Google group I had set up. Pro tip: give your language a made up name and you can secure all the 'places' and search engine results relating to it. Someone posted the site link to reddit and Hacker News and the site got 30k hits in the first 3 days. The interest level was quite surprising, and I attribute it partially to Paul Graham's essays on Lisp of the time [Graham 2003].

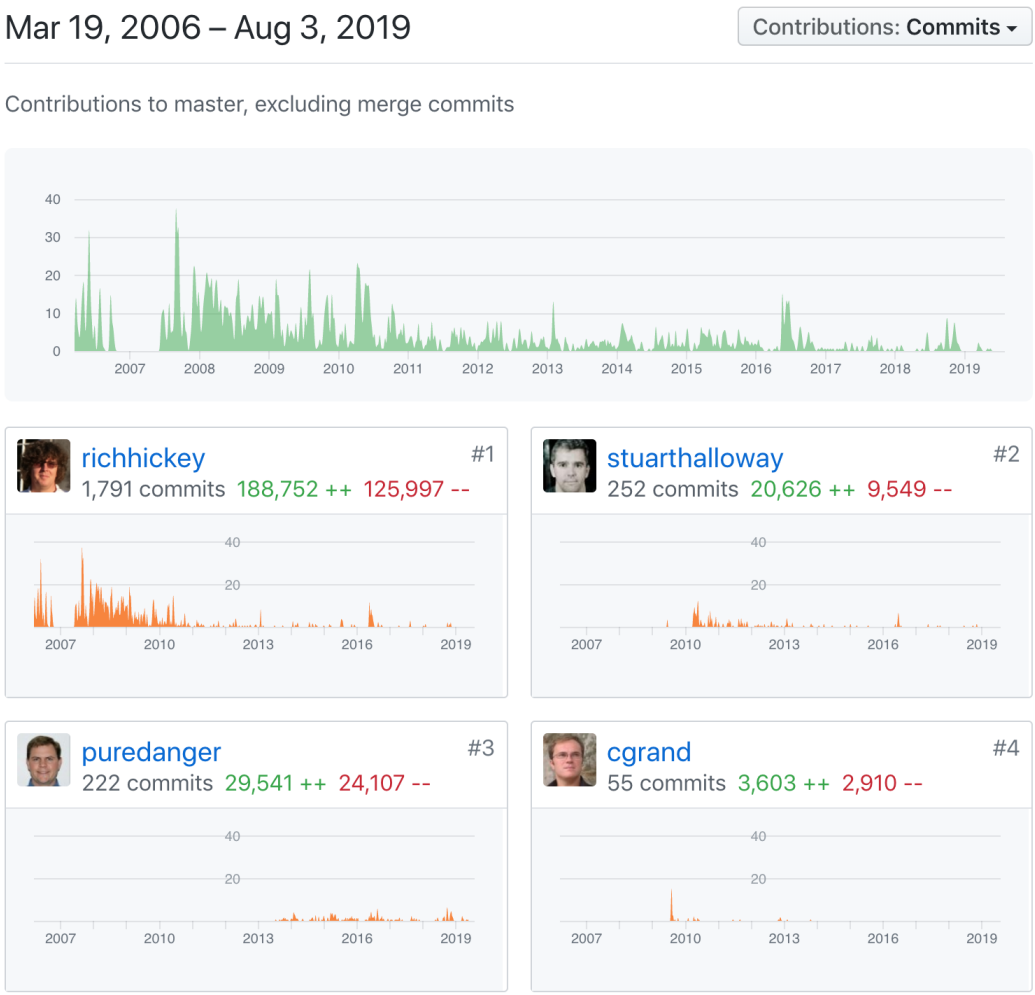


Fig. 2. Commit history in the Clojure repository, with top 4 committers

In November of 2007 I gave my first talk about Clojure to about a dozen people at the LispNYC user group, which I had been attending for several years. Several of the attendees of that "yet another hobby Lisp" talk have gone on to careers as professional Clojure programmers. That was the first of nine invited talks that I gave about Clojure or functional programming over the following year, including talks at ECOOP, IBM research, OOPSLA, QCON, and the JVM Language Summit, a speaking engagement pace I maintained or exceeded through 2015.

The 18 months following the initial release were a whirlwind of development. Many of the features documented above were refined in this period. Most substantially, the library of core Clojure functions (`clojure.core`) grew from about 100 to over 400. Thus, without more language constructs, the *vocabulary* of Clojure grew dramatically. This was significantly driven by the needs of the first users.

A #clojure IRC channel was started and I monitored it attentively [ClojureIRC 2008], talking to users, refining or adding functions to meet their needs, fixing bugs, answering questions, playing

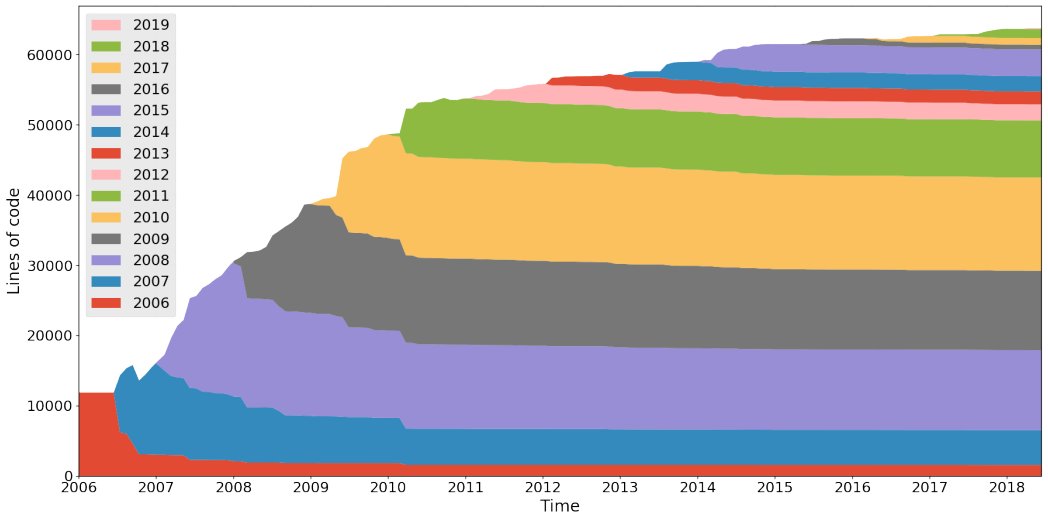


Fig. 3. Clojure codebase—Introduction and retention of code

the 'naming game' for new functions etc. IRC conversations with Chris Houser, Christophe Grand, Aaron Brooks, Chas Emerick and others were invaluable in helping me learn how to explain Clojure, understand the challenges of users, and formulate new features.

My brother, Tom Hickey, designed the Clojure logo. The Google group mailing list approached and crossed 100 users and I joked with my wife that I might eventually get to 500 users, to which she replied "don't get your hopes up". The list now has over 11,000 members, even though conversations about Clojure now occur in many forums, including the clojurians Slack (over 16,000 members) and Clojure subreddit (over 17,000 members).

Users were able to follow Clojure development in the sourceforge svn. Quite frequently users would find a bug or have a need, I would code it up within minutes, let them know in IRC that it was present in 'revision nnn', and they would pull that and give me feedback. Thus early users were consistently helping me test code between 'releases', which at that time were just the zipping up of the repo at a particular date. I issued such alpha 'releases' in June, September and December of 2008.

In this time frame I began to accept patches from users. Of the 200k lines of code in Clojure 1.0, roughly 3% came from ten contributors. Stuart Halloway helped with assessing and applying patches, the largest of which was a testing framework authored by Stuart Sierra. All coordination happened via mailing lists and internet-hosted repositories.

I released Clojure 1.0 in May of 2009, and at that time the first book on Clojure [Halloway 2009] was released. That book is now in its third edition, and code from the first edition still works. There are now dozens of books about Clojure.

I never returned from my 'sabbatical'.

4.2 December 2009, Clojure 1.1

Clojure 1.1 brought the chunked sequences and transients, a system for fast, single-threaded, mutating construction of the persistent data structures, with O(1) conversion to their immutable state. The number of contributors grew to twenty, with the majority of patches being bug or typo fixes of fewer than 50 lines. Christophe Grand helped on the data structures and transients, Chris

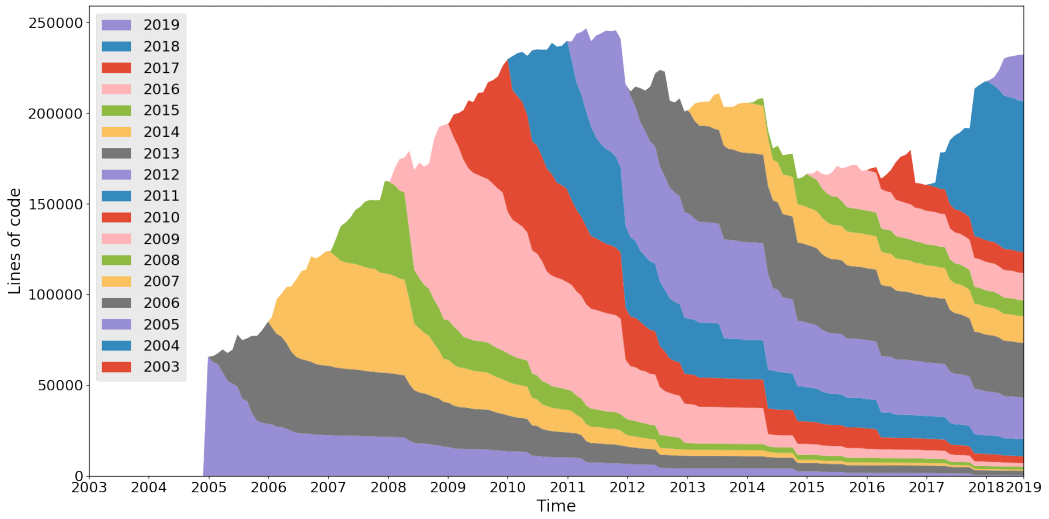


Fig. 4. Scala codebase—Introduction and retention of code

Houser helped with chunked sequences and the `for` syntax macro, and Timothy Pratley worked on support for primitive arrays.

There were some deprecations and removals in 1.1, and this was pretty much the last of that. Moving forward, Clojure was to be extremely conservative about breaking changes. I wanted Clojure to be a stable tool for professionals, not a platform for experimentation.

Figure 3 (created with the Hercules repository analysis tool [source d 2020]) shows when code was introduced in the Clojure repository and for how long it has been retained. This captures both the nature of the code being written and the approach to stewardship on the project. The drop of 2006 reflects pre-release bulk renaming and reorganizing of the code.

As a point of contrast, and without criticism, consider figure 4, the same graph for the popular programming language Scala. Interpretation of the differences is nuanced - whether a high rate of change is better or worse is a question for the consumer. Notably absent from Clojure survey results are complaints about stability or breaking changes.

In February 2009, David Miller of DePaul University started a port of Clojure back to the CLR, porting the Java code to C#. ClojureCLR was taken under the umbrella of Clojure stewardship, which included at this point many other Clojure 'contrib' libraries. The contrib libraries were covered by Clojure's open source license and contributor's agreement, and covered many functional areas not covered by the core library, like SQL interfacing, combinatorics, JSON and XML processing, string utilities, etc. Contrib has grown to cover 44 projects, with 348 contributors who've released 860 versions of the libraries over the years.

Even though Clojure was released under an open source license (the CPL, which became the Eclipse Public License—EPL), I always also have required a signed contributor's agreement from anyone making any contributions to Clojure itself or the contrib libraries under its umbrella. This ensures that, should the winds change on which open source license is best, the entire codebase can be moved to another license without having to find all participants and get their agreement.

In December 2009, having stretched my self-funding for almost 4 years and depleted my retirement savings, I conducted a funding drive to continue development for 2010. Several companies and many individuals participated.

4.3 2010, Clojure 1.2—Polymorphism Revisited

The summer of 2010 brought Clojure 1.2, which contained the most significant as-yet-missing features of the language: `protocols`, `reify` and `deftype/defrecord`. Until this point Clojure had been missing an ability to define high-performance runtime-polymorphic functions. In order to get the desired performance, the mechanism had to eventually turn into the mechanism of the host. On the other hand, I didn't want the host semantics (e.g., inheritance) to become Clojure's. Finally I wanted an open system, like Common Lisp generic functions [Gabriel et al. 1987] or Haskell's type classes [Wadler and Blott 1989].

The solution I came up with was *protocols*, named sets of polymorphic functions. One way to look at them is as a named set of single dispatch generic functions, another is like dynamic type classes. `defprotocol` defines the signature and documentation of the functions but no implementation. With a new construct, `reify`, programmers can construct an anonymous object that satisfies the protocol, similar to the way they could get an anonymous implementation of a Java interface with the interop feature `proxy`, and in fact `reify` also covers the interface implementation case, with improved performance, and abstracts away the differences.

With `defrecord` programmers can define new maps that include the necessary internal type tagging to support protocols, define protocol implementations inline, and provide positional constructors and other features. `extend-type` lets a single definition add support for one or more protocols to an existing record or host type, and `extend-protocol` lets a single definition extend a protocol to one or more existing records or host types.

```
;; Counted is a protocol containing one function - cnt
(defprotocol Counted
  (cnt [_] "how many?")
  ;; there could be more functions here...
  )

;; reify can make an implementation of a protocol on the fly
;; given definition(s) of the protocol's method(s)
;; the first argument to a protocol fn is the implementing object (analogous to Java's 'this')
;; here we don't use it, so we call it '_'
=> (cnt (reify Counted (cnt [_] 17)))
17

;; defrecord defines a new map-like type that can satisfy one or more protocols
(defrecord MyCounted [x]
  Counted
  (cnt [_] x)
  ;; more protocols here...
  )

;; defrecord also defines a constructor function '->record-name'
(def mc (->MyCounted 42))

=> mc
#user.MyCounted{:x 42}

=> (cnt mc)
42
```

```
;; records _are_ maps
=> (assoc mc :new :stuff)
#user.MyCounted{:x 42, :new :stuff}

=> (keys (assoc mc :new :stuff))
(:x :new)

;; extend-protocol connects a protocol to type(s)/records(s) we don't control
;; here providing an implementation of the Counted protocol for Java's String class
(extend-protocol Counted
  java.lang.String
  (cnt [s] (.length s))
  ;; implementations for more types here...
  )

=> (cnt "foo")
3

;; or extend a single type to protocol(s), same result
(extend-type java.lang.Long
  Counted
  (cnt [n] n) ;;dubious semantics here
  ;; more protocols here...
  )

;; open runtime polymorphism - mapping cnt across a heterogeneous list
=> (map cnt ["abc" 123 mc])
(3 123 42)
```

Under the hood, protocols define Java interfaces, and `deftype`/`defrecord`s that have inline protocol definitions establish an inheritance relationship. At an invocation point of a protocol, the compiler creates a call site with a fast path for interface implementations and a polymorphic inline cache for types externally extended to satisfy the protocol. In this way, if programmers use either inline definition in `deftype` or `defrecord`, or use `reify`, they get performance on par with Java's native polymorphic dispatch, and otherwise the performance is still good.

`deftype` is like `defrecord` but does not generate support for the map abstraction, while it *does* support mutable fields etc. It is for bottom-level plumbing tasks.

With these features Clojure programmers were equipped with the tools to define data structures and other constructs with polymorphic performance on par with Clojure itself. (Re)writing Clojure in Clojure would still wait, until ClojureScript.

In 2010 we moved the development system to Jira and Confluence, after using Assembla for a little over a year. These issue tracking systems allowed users to report bugs, propose fixes, and, having signed the contributor's agreement, submit patches. Stuart Halloway helped me in assessing and applying patches. All changes to Clojure were and are individually reviewed and approved by me.

Clojure 1.2 had 28 contributors. The significant non-fix contribution was an implementation of Common Lisp-like `format`/`pprint` from Tom Faulhaber.

In the fall of 2010 we held the first Clojure conference—the Conj. When we asked "who's using it for work" only a few hands went up. The Conj has been run annually since, supplemented by ClojureWest, EuroClojure, clojureD, ClojuTRE, Clojure/South and others. When we ask now "who's using it for work?" almost everyone raises their hands.

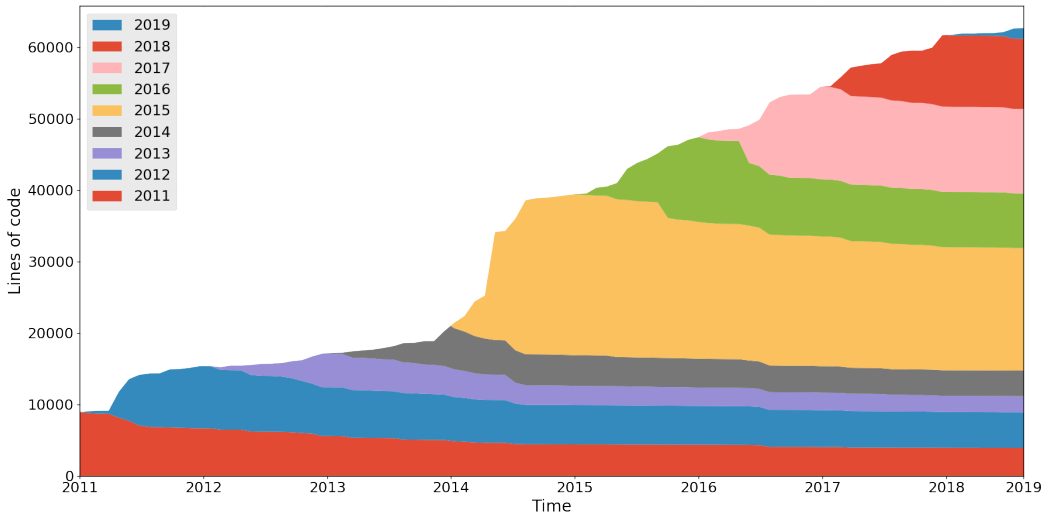


Fig. 5. ClojureScript codebase—Introduction and retention of code

4.4 2011, ClojureScript

In 2011 I started work on ClojureScript, an implementation of Clojure that ran on JavaScript (JS) engines (e.g., browsers, NodeJS). The idea was to combine Clojure's programming model with JavaScript's reach, JavaScript being the only natively supported language in browsers. I wrote the first ClojureScript compiler in Clojure, and it compiled Clojure code to JavaScript. A key decision I made early on was to ensure that the JavaScript produced would be acceptable to the advanced compilation mode of Google's Closure (yes, homophone) compiler, a JavaScript to JavaScript compiler supporting advanced optimization including dead code elimination. This is important for a language like Clojure where developers might require a large library only to use some small subset of it, and minimizing (not just minifying) the code transmitted to browsers is critical [McGranaghan 2011]. This choice underlay the other objective: ClojureScript, like Clojure, is meant for production programming, with a similar approach to stewardship (figure 5).

Shortly after getting the proof of concept started I expanded the team to some coworkers and a few Clojure community members, among them David Nolen, who went on to become the development lead and primary steward of ClojureScript. With a handshake over lunch David agreed that ClojureScript would be Clojure on JavaScript, and never morph into a variant or dialect, and he has done an excellent job since making sure that is so, as well as bringing his considerable expertise in JavaScript to bear.

The implementation of ClojureScript was a real test of Clojure's design and abstractions. Here was a new host platform that I did not have in mind when I first wrote Clojure. The object model was significantly different from Java/C#: prototype vs class and inheritance based. The numerical types were different. Methods are just functions in fields etc. Fast polymorphic dispatch is supported but has a significantly different implementation.

The ClojureScript implementation was to be Clojure-in-Clojure. I started by building the foundations for `deftype` and `defprotocol`, and then the team built the abstractions and data structures, followed by the core library in ClojureScript, on Clojure constructs, rather than on JavaScript, thus differing from the way Clojure was built on Java. Other than the need to introduce `.-field` accessor support, the existing host syntax mapped well to JavaScript. `defprotocol`, `reify`, `defrecord` and

deftype were sufficiently abstracted away from their Java implementations. Protocols were fast enough to be used to build the abstractions at the bottom on the core library.

ClojureScript is Clojure — it is not a subset, a 'lite' version or a look-alike. ClojureScript supports all of the constructs of Clojure with the same semantics, all of the core libraries, and many large ported libraries. Programmers can print/read to/from Clojure and ClojureScript, etc. Many production Clojure shops use ClojureScript on the front end. ClojureScript's interactive development tooling is best-of-breed, and Clojure's immutable data structures have been a surprising performance win when dealing with UI frameworks (such as React) that require fast structural diffs.

In short, Clojure, via ClojureScript, became a compelling language for web development.

4.5 2012, Clojure 1.4

The signature feature in 1.4 was the introductions of tagged literals to the reader, and the specification of 'edn'—extensible data notation [edn 2014], the data subset of the Clojure read/print format.

edn supports extensibility through a simple mechanism. # followed immediately by a symbol starting with an alphabetic character indicates that symbol is a tag. A tag indicates the semantic interpretation of the following element. It was envisioned that a reader implementation will allow clients to register handlers for specific tags. Upon encountering a tag, the reader will first read the next element (which may itself be or comprise other tagged elements), then pass the result to the corresponding handler for further interpretation, and the result of the handler will be the data value yielded by the tag plus tagged element, i.e. reading a tag and tagged element yields one value. This value is the value to be returned to the program and is not further interpreted as edn data by the reader.

This process will bottom out on elements either understood or built-in. Thus programmers can build new distinct readable elements out of (and only out of) other readable elements, keeping extenders and extension consumers out of the text business.

The semantics of a tag, the type and interpretation of the tagged element, are defined by the steward of the tag, and the spec requires that user extensions use namespace-qualified, and thus conflict-free, symbols as tags, reserving unqualified symbols for the standard. Two such tags were included:

```
;; An instant in time.
;; The tagged element is a string in RFC-3339 format.
#inst "1985-04-12T23:20:50.52Z"

;; A UUID.
;; The tagged element is a canonical UUID string representation.
#uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6"
```

When read in Clojure on the JVM, they produce the expected types:

```
=> (class #inst "1985-04-12T23:20:50.52Z")
java.util.Date

=> (class #uuid "f81d4fae-7dec-11d0-a765-00a0c91e6bf6")
java.util.UUID
```

The critical property of this system is that any edn reader, without enhancement, can read any edn file. Specifically, it can read tagged literals it doesn't understand (i.e. for which there is no handler), know that it doesn't understand them, and allow for programmer-defined policy in that

case (e.g., failure, skip, generic handling). Thus it is possible to make transformers and conveyance mechanisms that need not understand *any* tags. This encourages the development of loosely coupled systems.

While tagged literals don't support the diversity that CL-style reader macros do, they do support many important use cases, such as the print/read of records, in an extensible way compatible with library development.

`edn` does not include any of the code-related reader transformations of Clojure's code reader, such as the conversion of `@x` into `(deref x)`. At present, it also does not support metadata, as it is a facility supported by few languages other than Clojure. However, the use of `edn` for Clojure-to-Clojure(Script) interop is widespread and metadata support would be desirable to fully leverage Clojure's data design.

4.6 2012, Datomic

In 2010 I started work on a new database that was released as Datomic in 2012. Databases are the giant mutable global variable at the heart of any system, and no amount of programming in functional languages is going to make a system with an update-in-place database much easier to reason about. My idea was to build a functional database that could present to the developer the entire database at any point in time as an immutable value. It would never update in place, nor have any notion of place. The data model would be akin to RDF, entity/attribute/value, with the addition of time. This, it seemed to me, was the final piece needed to get us Out of the Tar Pit [Moseley and Marks 2006].

I implemented Datomic, with Stuart Halloway and others, in Clojure. Datomic was a good example of the kind of server program I used to develop prior to working on Clojure, and Clojure's features greatly facilitated its rapid development. Interop was critical to integrating with the many communication and storage APIs we wanted to support. I designed the query language, an implementation of Datalog, as a data-based language, like Clojure, and got a runtime query compiler for free by leveraging the Clojure compiler. Transaction data and queries are all just data expressible as edn, and thus programmable, read/printable, etc. Clojure's concurrency support made for a worry-free multithreaded architecture. In my estimation, the codebase would have been several times as large had we developed it in Java.

Datomic is a commercial product, and its revenues help pay for ongoing Clojure development.

4.7 2013, Cognitect

In 2013 we formed Cognitect, a merger of Metadata Partners, a company we formed around the development of Datomic, and Relevance, a consultancy founded by Stuart Halloway and Justin Gehtland that had come to focus on Clojure. Cognitect eventually hired Alex Miller to work almost full-time on Clojure. In providing a livelihood for the core team, Cognitect became, and remains, the de facto corporate sponsor of Clojure. Cognitect is the vehicle through which we organize and fund the Conj as well as various other Clojure community outreach programs.

At Cognitect we solidified the process for maintaining Clojure. Alex Miller vets and refines initial bug and feature requests, I review and approve those for implementation, Alex and Stuart Halloway screen the resulting patches, whether implemented by us or others, and I review the patches for final inclusion. Major features were, and continue to be, initiated and designed by me, while recently Alex has taken on an increasing portion of the implementation, at my direction.

4.8 2013, core.async

In 2013 I designed an implementation of CSP [Hoare 1978] akin to that provided by the Go language, as a library for Clojure called `core.async`. This provided an important portable abstraction of

communication over channels to Clojure and ClojureScript. On the ClojureScript side it provided a significant improvement over "callback hell".

4.9 2014, ClojureBridge

In April 2014 Cognitect hosted the first ClojureBridge workshop. ClojureBridge is an all-volunteer organization dedicated to increasing diversity within the Clojure community by offering free, beginner-friendly Clojure programming workshops for underrepresented groups in tech. They have since gone one to host over 75 workshops in cities around the world [clojurebridge.org 2020].

4.10 2015, Clojure 1.7

Many refinements were made in the intervening releases, but Clojure 1.7 included two significant features: *reader conditionals* and *transducers*. Reader conditionals allowed mostly-portable code to conditionally include some host-specific code, with compile-time detection of target host and inclusion. This and other features of the release substantially improved the ability of library developers to simultaneously target Clojure and ClojureScript from a single codebase.

Transducers are composable algorithmic transformations. In working on enhancements to `core.async`, it was natural to want versions of `map`, `filter`, etc that worked on channels. Similar facilities are provided by other "push" oriented and reactive systems. However, while recognizing and talking about duality might be fun, implementing everything over and over again is not. The industry practice of implementing `map` et al, and the composition thereof, for collections, then for streams, then for observables, then for channels is redundant, too specific and inefficient. So I set out to see if there was a way to write `map`, `filter` et al once and for all.

The basic idea is to leverage the universality of `fold` [Bird et al. 1988], but to take the list out of `fold`. Here's a short derivation:

```
;; We can define map, filter etc with reduce (Clojure's foldl)
;; conj adds to the end (right) of vectors []
(defn map-left [f coll]
  (reduce (fn [r x] (conj r (f x)))
          [] coll))

(defn filter-left [pred coll]
  (reduce (fn [r x] (if (pred x) (conj r x) r))
          [] coll))
```

The basic problem is the presence of `conj` in the reducing functions; it complects the essential process transformation (mapping, filtering) with one specific concrete process (list/vector building). We want to get rid of `conj`. So let's turn it into an argument:

```
;; a transducer is a fn of reducing fn -> reducing fn
(defn mapping-transducer [f]
  (fn [step] (fn [r x] (step r (f x)))))

(defn filtering-transducer [pred]
  (fn [step] (fn [r x] (if (pred x) (step r x) r))))

;; redefine with transducers
(defn map-left [f coll]
  (reduce ((mapping-transducer f) conj)
          [] coll))
```

```
(defn filter-left [pred coll]
  (reduce ((filtering-transducer pred) conj)
          [] coll))
```

The critical thing to note is that the `step` function can be *anything*, including a side-effecting, active process like informing observers, firing missiles, or putting something on a channel. Transducers compose via ordinary function composition. This means one can construct and compose transformation pipelines without knowledge or concern about the ultimate context of their use. And they are efficient, incremental, inline-able, allocation minimal etc, obviating the need for a sufficiently smart compiler, laziness and stream fusion. The actual transducers are somewhat more involved than above, e.g., they support early termination.

In 1.7 we added an additional arity to all sequence functions for which a transducer made sense, such arity being one lower than the 'normal', collection-taking arity, and had it return the corresponding transducer. Such things are possible because Clojure supports variadic functions and heterogeneous returns. Thus:

```
(map f coll) ;; => transformed sequence
(map f) ;; => mapping transducer
```

We modified core.async channels to accept and use a transducer, and we didn't, and won't ever again, have to rewrite `map` et al.

```
;; in action
;; transducers compose with ordinary function composition - 'comp'
=> (let [xform (comp (map inc) (filter even?))
         achan (clojure.core.async/chan 1 xform)]
     ;; achan will increment and filter everything put on it
     (do-something-with achan)
     ;;  use the exact same transducer pipeline for sequence reduction
     (reduce  (xform conj) [] (range 10)))
[2 4 6 8 10]
```

Thus we tweak Perlis one final time: "It is best to have 100 functions operate on NO data structure".

I think transducers are a fundamental primitive that decouples critical logic from list/sequence processing and construction, and if I had Clojure to do all over I would put them at the bottom. Transducers have since been implemented by users of many programming languages. [Björnstam 2019], [Puente 2015].

4.11 2016—2019, Clojure 1.8—1.10

Subsequent releases of Clojure focused on bug fixes, improved error handling and other refinements. Nicola Mometto, Andy Fingerhut and others consistently help out in assessing and supplying patches. Alex Miller has worked on command line tooling and installation support. The next prominent language feature I am working on, still in development, is `clojure.spec`, a system that combines predicative specification of data structures and functions with data generation and generative testing.

Outside of the core language, the Clojure ecosystem is continuously growing. There are over 32,000 GitHub repositories tagged 'Clojure'. Ambrose Bonnaire-Sergeant has done interesting research work on Typed Clojure, an optional type system for Clojure. [Bonnaire-Sergeant et al. 2016].

5 RETROSPECTIVE

5.1 Community

Having not ceded to *readability* or familiarity pressures, Clojure was never destined to unseat Java. And yet there are now estimated upwards of 50,000 Clojure programmers, two thirds of which in surveys indicate they use it at work (table 3). Hundreds of companies use Clojure, several of which employ more than 100 Clojure programmers. One can have a career as a Clojure programmer.

Yet Clojure's adopters have to overcome many personal and organizational hurdles in order to succeed.

Most developers come to Clojure from Java, JavaScript, Python, Ruby and other OO languages. The most commonly anticipated problem is unfamiliar syntax, "dealing with all those parentheses", but the most significant actual problem is learning functional programming. Clojure is not multi-paradigm, it is FP or nothing. None of the imperative techniques they are used to are available. That said, the language is small and the data structure set evident. Clojure has a reputation for being *opinionated*, opinionated languages being those that somewhat force a particular development style or strategy, which I will graciously accept as meaning the idioms are clear, and somewhat inescapable.

While there is a learning curve, users experience benefits all along the way. They find the amount of code they have to write is significantly reduced, 2—5x or more. A much higher percentage of the code they are writing is related to their problem domain. Their defect rates are reduced. And they gradually come to realize how much of their mental energy previously had been consumed by trying to manage state. There are many testimonials to these benefits interspersed among the conference talks on ClojureTV [2019].

Not all is rosy. Users perennially report dissatisfaction with the error reporting from the compiler and various macros. They are confused by Java's stack traces. They face challenges maintaining and understanding, e.g., the data requirements of a codebase when a discipline around documentation has not been maintained. If they have not already programmed in Java, they find the need to understand Java in order to leverage host interop an additional learning curve with which Clojure provides no assistance. These remain ripe areas for improvement of Clojure and work proceeds to address them on several fronts.

While Clojure's hosted-ness is of great benefit in greasing the skids of adoption, the normal institutional inertia and pressures exist. Who else is using this? Where will we find programmers? How will we train them? Only users' success with the language and the continued growth of the community can assuage these concerns.

Ultimately, the adopting developer and organization sits in and relies upon a cradle of support created by the community surrounding a language. And it is one of the highlights of the Clojure experience for me that the community surrounding Clojure is terrific. When I first was learning Common Lisp I was appalled at the tolerance on Usenet `comp.lang.lisp` of arrogance, hostility, intimidation and other anti-social behavior of the presumed 'smartest guys in the room'. I was determined (a) that the Clojure community would not be hosted there and (b) that kind of behavior would not be tolerated. Early on, on IRC and the mailing list, I established and required a tone of supportiveness, positivity and respect. That has since been seized upon, maintained and enforced by the community itself as one of their core values and points of pride; the Clojure community is friendly and helpful and tolerates nothing less. While this can't be substantiated except by direct experience, it remains true and important nonetheless.

5.2 Evangelism

One of the many unanticipated results of releasing Clojure was getting many invitations to give talks. Initially I was asked to talk about Clojure, but increasingly I was being asked to give keynotes. In all cases, but especially in the keynotes, I endeavored to talk about the ideas behind Clojure, or functional programming. I never 'sold' Clojure, i.e. encouraged someone to use it. I would just explain what I saw as the benefits, perhaps enthusiastically, but believed there was no point trying to convince someone who wasn't already inclined or identifying with the problems Clojure addressed.

Five years of travel, much of it international, took its toll though. Prior to releasing Clojure I had rarely traveled, for work or otherwise. And it was not just the travel; preparing talks, especially keynotes, was a lot of work. There is no doubt though that the talks, and the fact that they subsequently were made available online, were a major factor in people becoming interested in Clojure. Simple Made Easy [Hickey 2011], a talk I gave at the Strange Loop conference in 2011, still attracts people to Clojure eight years later.

5.3 Stewardship

Another unanticipated outcome of Clojure taking off was the responsibility for stewarding a community. When writing Clojure I never expected more than a handful of users, as I had for my other projects like jFli. Soon there were dozens, then hundreds, then thousands of people asking questions, looking for clarifications and guidance, and most challengingly, desiring input into the project.

When I open sourced Clojure, what I thought I was doing was sharing something I had created in a way, open source, that would provide no barriers to adoption. What I discovered was that open source engenders presumptions of collaborative development, which can be at odds with maintaining a singularity of vision. For good and bad, I have prioritized maintaining my vision for Clojure and have been self-protective about the coordination demands of stewardship dominating my time and energy. This has, at times, caused friction with those in the community who think open source should be run a certain way. The stresses of stewardship have been greatly eased by the participation of my co-workers at Cognitect, Stuart Halloway and Alex Miller.

Another presumption in open source is a high rate of change, as problems are fixed, things 'improved', misfeatures deprecated and removed, and the latest ideas pursued. I considered this approach antithetical to commercial adoption, especially at the language level. I took Clojure being at the bottom of programs written in it very seriously — Clojure could not be a lab for experimentation. Breaking change was and is considered extremely bad, and it is strenuously avoided in Clojure. So I took my lumps and lived with my decisions, and users of Clojure benefitted from ongoing stability. Clojure exhibited a maturity in that regard that belied its age, which helped me build trust with early adopters. Again, this was to the consternation of some in the community who wanted the more familiar open source experience.

5.4 If Only...

I should have considered more the experience of people who 'get it wrong' and the importance of good error messages in setting them right, and prioritized that early on. In jest, I blame it on my background coming up programming in C, where when we got it wrong our programs just crashed and we stared at the source code until we figured out why. As character-building as that was, it need not be so today.

Clojure always involved compromises in pursuit of practicality and I am fine with that. Clojure is an exercise in tool building and nothing more. I do wish I had thought of some things in a different

order, especially transducers. I also wish I had thought of protocols sooner, so that more of Clojure's abstractions could have been built atop them rather than Java interfaces.

Clojure had no institutional sponsor, and I definitely worked on the project without income for too long. We'll see how that works out come retirement time.

When I embarked on Clojure I was not, nor ever aspired to be, a language designer and steward, and from now on I am responsible for Clojure. The success of Clojure has changed my life and career in ways I could never have anticipated.

6 ADOPTION AND USER SUCCESS

Ultimately the success of a language rides upon the success of its users. Whatever design or evangelism we do, users telling their stories, a process not under our control, is what causes a language to spread.

6.1 Surveys

Annually, since 2010, we have surveyed the Clojure user community in an effort to better understand how and why they use Clojure. The results from 2010 (table 2, 487 respondents) show a nascent community of interested users, primarily coming from OO backgrounds, fewer than 25% of which are using it for work.

A free response question in 2010 regarding Clojure weaknesses had many responses indicating:

- Poor error messages and incomprehensible stack traces
- Documentation not helpful enough
- Getting started difficulty
- Lack of IDE support

The most recent 2019 Clojure survey (table 3, 2,461 respondents) shows an established community with many experienced users, still attracting newcomers from OO languages. A substantial number of respondents use it at work. Also interesting is the penetration of ClojureScript amongst Clojure shops—fully 60% are using it in addition to Clojure, presumably for the front-ends of their systems.

Figure 6 demonstrates a substantial rise in the use of Clojure for commercial and enterprise applications. Figure 7 shows the triumvirate of functional programming, the Lisp REPL-driven development experience, and host platform access are most valued. Figure 8 shows the ongoing challenges of adoption: internal evangelism and hiring.

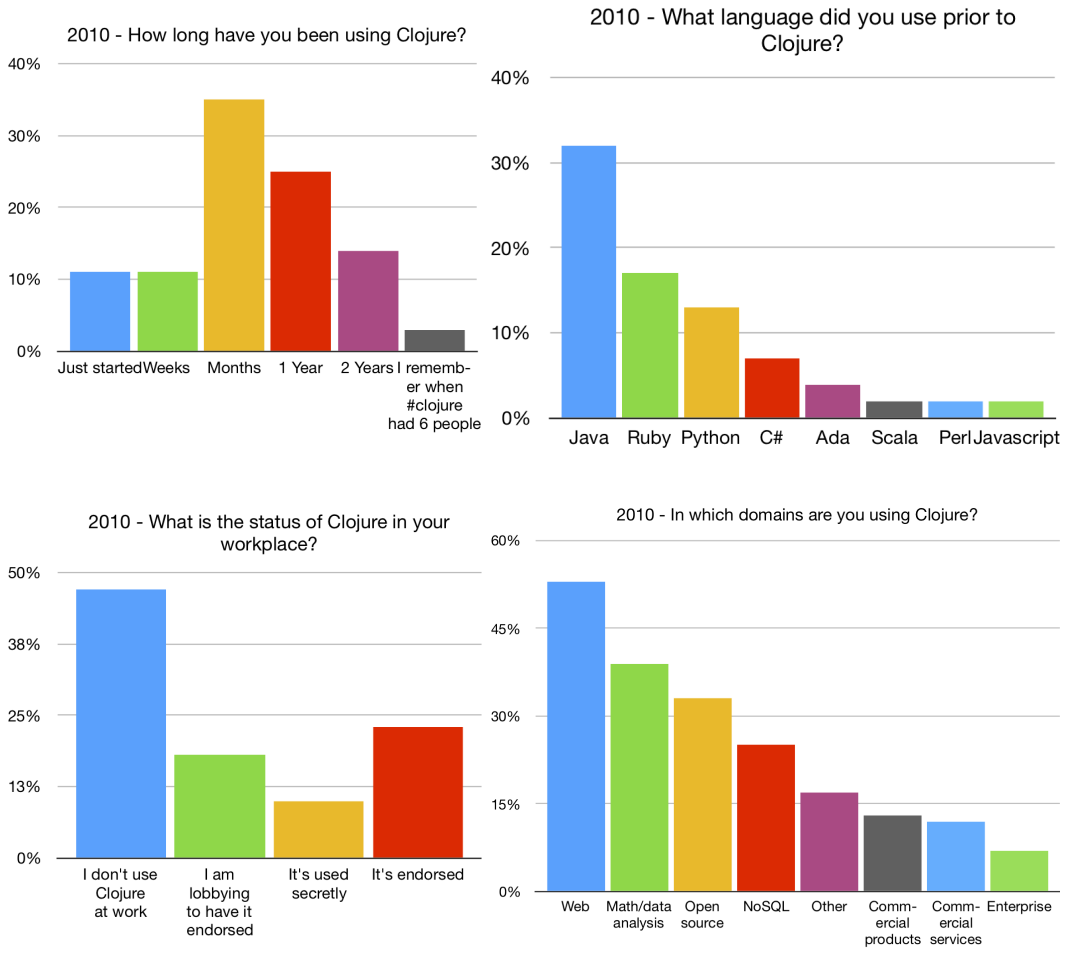
StackOverflow's 2019 Developer Survey [Overflow 2019] (90,000 participants, figure 9), revealed that Clojure programmers are among the highest paid and most experienced professionals in the industry.

6.2 Success Stories

Clojure is in production at hundreds of companies. The clojure.org site lists over 400 companies that have publicized their use [clojure.org 2019]. Many companies have provided case studies and users given conference talks about their experience with Clojure. Here are some excerpts.

6.2.1 Anglican—a Probabilistic Programming System. "Anglican [Tolpin et al. 2016] is a probabilistic programming language integrated with Clojure and ClojureScript. While Anglican incorporates a sophisticated theoretical background that you are invited to explore, its value proposition is to allow intuitive modeling in a stochastic environment. It is not an academic exercise, but a practical everyday machine learning tool that makes probabilistic reasoning effective for you. Anglican and Clojure share a common syntax, and can be invoked from each other. This allows Anglican programs to make use of a rich set of libraries written in both Clojure and Java. Conversely Anglican

Table 2. 2010 Clojure survey results

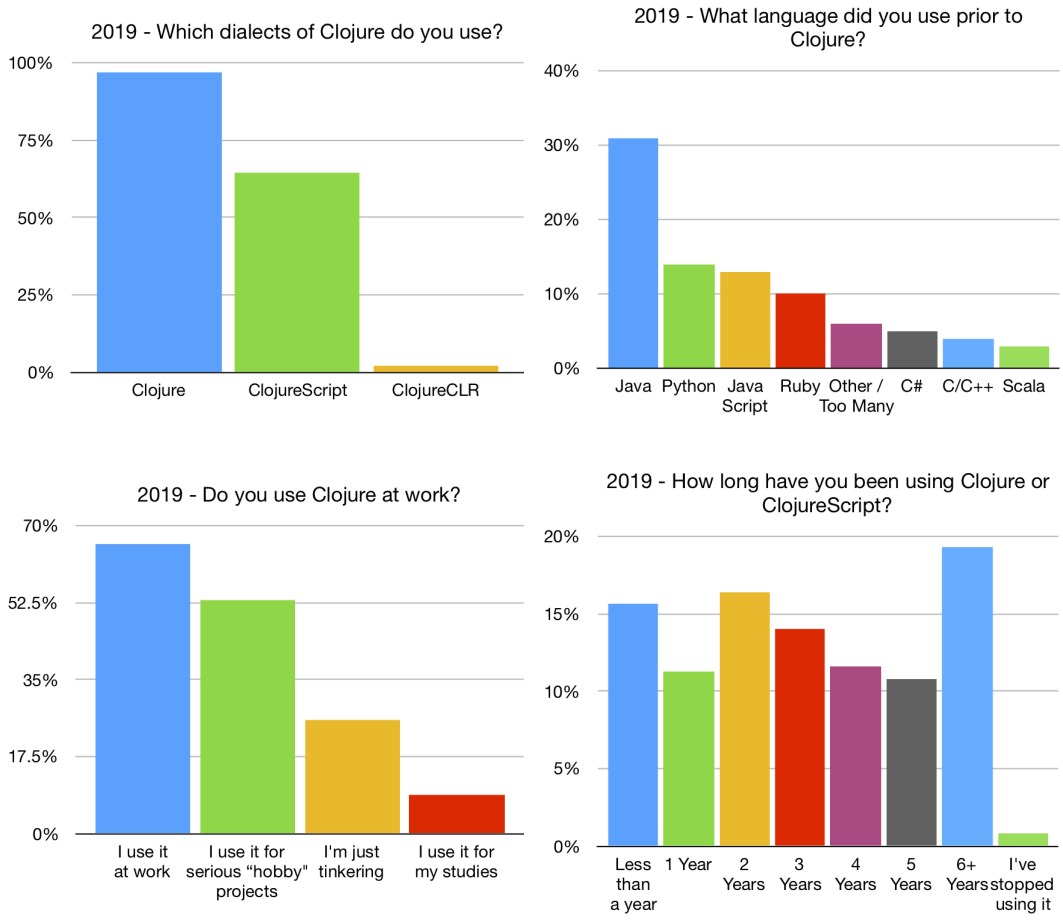


allows intuitive and compact specification of models for which inference may be performed as part of a larger Clojure project." [Anglican 2020]

6.2.2 Walmart—eReceipts. In preparation for the 2014 holiday season, project architect Anthony Marcar's team at WalmartLabs rolled out a new system designed from the ground up to handle the projected shopping frenzy. Walmart's eReceipts team of eight developers used Clojure "all the way down" to build a system to process and integrate every purchase at Walmart's 5000+ stores, including online and mobile data. The data driven system enabled the popular Black Friday 1-hour guarantee and other programs to improve the customer experience and streamline operations. After the staggering wave ebbed, he tweeted: "Our Clojure system just handled its first Walmart black Friday and came out without a scratch". [Marcar 2015]

6.2.3 Netflix—Mantis Query Language. "Among other things at Netflix, the Mantis Query Language (MQL an SQL for streaming data) which ferries around approximately 2 trillion events every day for operational analysis (SPS alerting, quality of experience metrics, debugging production, etc) is

Table 3. 2019 Clojure survey results



written entirely in Clojure. This runs in nearly every critical service, 3,000 autoscaling groups and easily > 100k servers. Clojure allows us to also compile it for our NodeJS services as well." [diab0lic 2018]

6.2.4 The Climate Corporation—Versioned Weather Datastore. The Climate Corporation (acquired by Monsanto in 2013) helps farmers around the world protect and improve their farming operations with powerful software, hardware, and insurance products. They combine hyper-local weather monitoring, agronomic modeling, and high-resolution weather simulations to deliver mobile SaaS solutions that help farmers make better-informed operating and financing decisions. They use Clojure for weather and crop models, managing data, and building services. "All our models, datastores, and services are built in Clojure. Among other things, Climate Corp built `mandoline`, a distributed, versioned, n-dimensional array database, in Clojure. We find that Clojure's support for parallelism makes it easy to run complex models with low latency." Leon Barrett—[clojure.org 2020]

6.2.5 Funding Circle—Peer-to-peer Lending Marketplace. "Funding Circle is a high profile peer-to-peer lending marketplace that allows investors to lend money directly to small and medium-sized

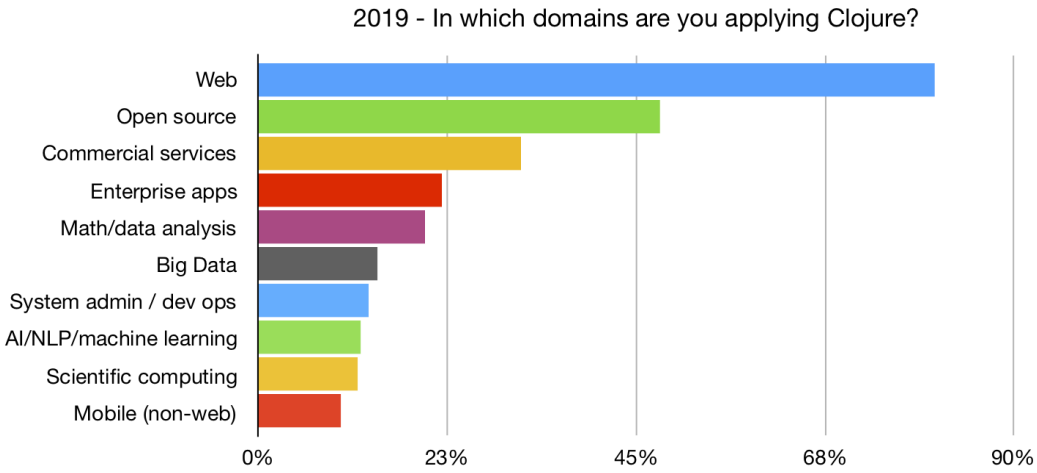


Fig. 6. 2019 Clojure survey—application domains

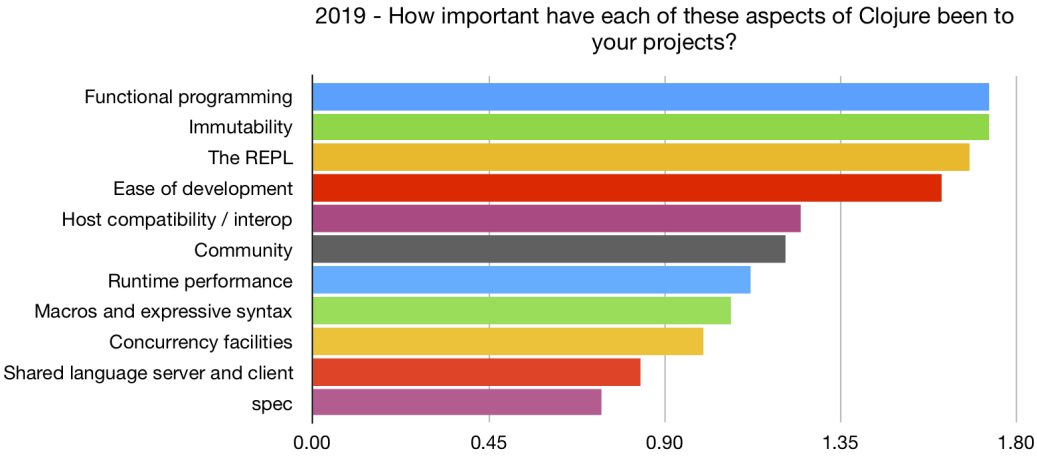


Fig. 7. 2019 Clojure survey—important features

businesses. Clojure is used extensively for the performance sensitive back-end services, for example making decisions on who to give loans to, the accounting system, and the handling of interactions with the upstream banks. The fact that Clojure is dynamic helps us bring our Ruby people across. There are also the pragmatic aspects of using the JVM; we're heavy Kafka users and there are native Java client libraries that we want to use. The approach we're taking is to 'turn the database inside out';using event streams to drive business logic over Kafka. Clojure is a good fit for this; mapping functions over unbounded streams. Clojure has held up its end and more. It's a fantastic language and the code is very maintainable." [Crim and Pither 2016]

6.2.6 HCA—Sepsis Prevention. "After being tasked with bringing data science to HCA, the largest hospital chain in the United States, we recognized the need for a large scale, life-saving project to demonstrate value and pave the way for getting the infrastructure and support that a meaningful data science operation would require. We used Clojure and Datomic to successfully deploy SPOT

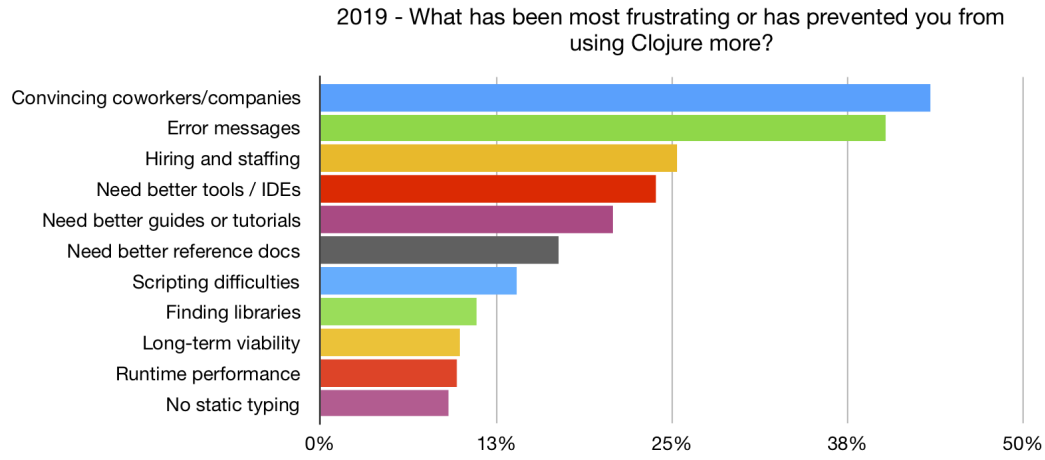


Fig. 8. 2019 Clojure survey—challenges

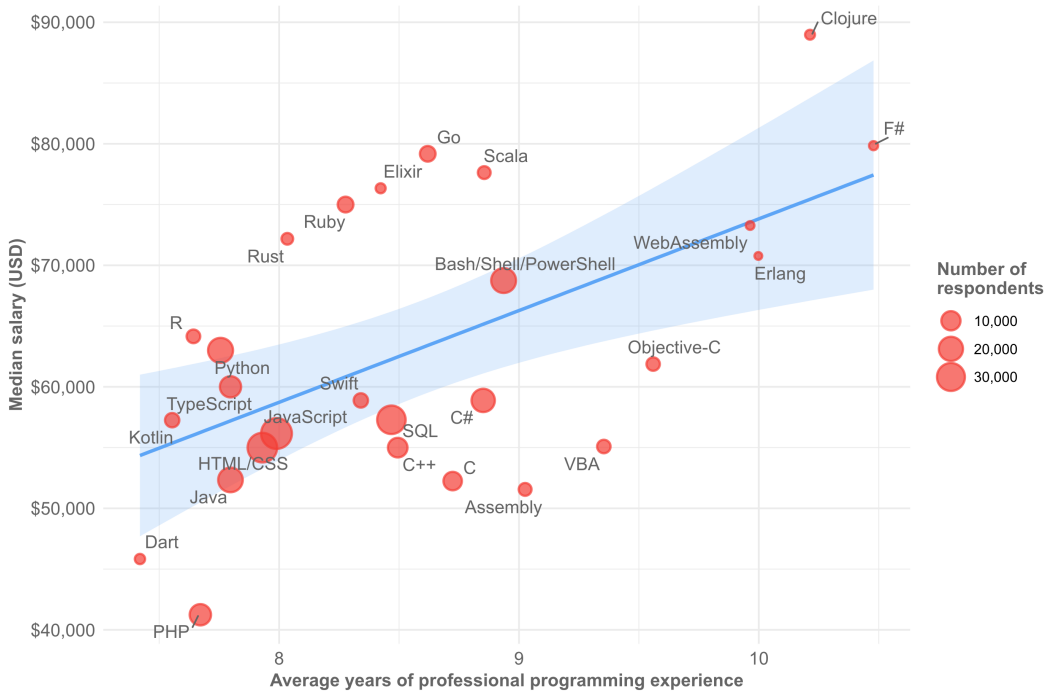


Fig. 9. 2019 Stack Overflow Developer Survey—salary and experience

(Sepsis Prevention and Optimization of Therapy), a real-time streaming clinical system, in 165 hospitals all across the US. Since SPOT launched, we've seen a dramatic increase in the percentage of sepsis patients receiving timely treatment, and we've seen an even more dramatic, and more important, decrease in severe sepsis mortality." [Ges and Castro 2018]

6.2.7 Polis—Technology Powered Democracy. "Polis is an AI powered surveying tool for large scale open ended feedback. At the heart of Polis is a machine learning system, powered by Clojure, which surfaces opinion groups and the comments important to them. Interactive data visualizations help groups better understand each other, and highlight points of consensus around which disputes can be resolved. Polis has been used to inform legislation and decision making in governments, companies and other organizations around the world. Polis chose Clojure for its support for concurrency, access to the JVM, and data driven approach to programming." [Small 2018] and private correspondence.

6.2.8 Sonian—Cloud Archiving. Adopting Clojure in 2009, Sonian was one of the first companies to use Clojure in production at scale. Their email archiving and search handles over 10-20 million email a day. All of their backend services are built in Clojure. "Clojure gives us a performance boost at both ends of the software development lifecycle: we can build new features as quick as any Python or Ruby shop, and when we ship it to production it runs as fast as any Java application." Ron Toland—[clojure.org 2020]

6.2.9 Nubank—a Bank Built on Clojure and Datomic. Nubank was started in 2013 with a credit card that was controlled through a mobile app. Since then, Nubank has expanded into additional financial services and currently has 850 employees in Brazil, over $1.1B in funding and 12 million credit card customers. There are currently over 250 Clojure developers in the company, and about 300 microservices written in Clojure. Nubank uses Datomic as its database. "We're a functional programming shop not in terms of really esoteric concepts, but just in terms of the basic immutability, idempotence, declarative format, small functions, pure functions. Clojure is an interesting choice because it's very simple. We're able to write actually quite a large system at this point without having any single piece of it that is difficult to understand." [Wible and Meyerson 2018]

7 CONCLUSION

To the extent Clojure was designed to allow professional programmers to write simpler, functional programs, at work, while building systems for as wide a variety of domains as are covered by Java and JavaScript, it has certainly succeeded, because that is precisely what the majority of its user community does with it.

I think Clojure distills important principles acquired from programming information systems in the large, especially those around achieving loose coupling via data-driven interfaces. This fosters a unique library ecosystem with a high degree of compatibility and composability, much higher than I've seen with library approaches that require the coordination of code-driven constructs.

Clojure embodies and demonstrates the continued viability of the Lisp idea, including the importance of a simple core, direct use of data, extension via libraries, and dynamic typing in delivering flexibility and reducing coupling. I would be happiest if Clojure were ultimately considered amongst languages like Common Lisp, Smalltalk and others that I perceive as being developed by people primarily occupied with *building systems*, and not (or not just) designing languages.

Clojure has introduced many professional programmers to the benefits of functional programming, and I am proud that Clojure, its community and I have been part of the steadily increasing dialog surrounding those benefits in the programming community at large.

Clojure targets real, experienced, costly programming problems like the complexity of state, the cognitive load of language features, the challenges of information, and coupling in systems. While I don't imagine Clojure's answers are definitive, I hope these problems and challenges continue to be a focus of system and language designers.

ACKNOWLEDGMENTS

My wife Stephanie supported my sabbatical, extended unemployment, career sidetrack and extensive travel, and listened for many, many hours to my ravings about persistent data structures, STM and other technical minutiae. Thanks Steph!

Among the many contributors to Clojure's and ClojureScript's development, codebase, conferences and contributed libraries were: Stuart Halloway, Alex Miller, Chris Houser, Michael Fogus, Eric Thorsen, David Nolen, Stuart Sierra, Christophe Grand, Tom Faulhaber, Chas Emerick, Tom Hickey, Justin Gehtland, Kim Foster, Phil Hagelberg, Andy Fingerhut, Nicola Mometto, Ambrose Bonnaire-Sergeant, Mike Fikes, António Nuno Monteiro, Steve Miner, David Miller, Brenton Ashworth, Michał Marczyk, Bobby Calderwood, Brandon Bloom, Timothy Baldridge, Ghadi Shayban, Kevin Downey, Daniel Solano Gómez, Alan Dipert, Sean Corfield, Meikel Brandmeyer, Reid Draper, Lynn Grogan, Gary Fredericks, Nada Amin, Mark Engelberg, Stephen C. Gilardi, Ben Smith-Mannschott, Luke Vanderhart, Aaron Bedra, Herwig Hochleitner, Nick Bailey, Alexander Taggart, Zach Tellman, Tassilo Horn, Devin Walters.

Stuart Halloway, Alex Miller and Michael Fogus assisted me by reviewing drafts of this paper.

A CODE AS DATA

Clojure shares with other Lisps the basic structure of programs as data (read from files or formed programmatically): nested lists where the first item in the list designates the 'operator' (s-expressions). The operator is usually a function, but can also be a macro or other invoke-able. To this code-as-data regime Clojure adds vectors `[]` and maps `{k v, ...}`.

Maps are rarely used as syntax, so if they occur in code they designate a data structure literal. Vectors however are uniformly used by syntax (macros) for grouping (of parameters, bindings etc), in preference over lists (). This means that an unquoted list in Clojure is invariably an expression, whereas in Scheme and Common Lisp the programmer must know the syntax of every macro in order to parse and understand its use of lists. This is a great example of where having more things is simpler than having one, semantically overloaded, thing. That said, vectors not subject to interpretation by macros are treated, like maps, as data literals.

```
;; this is a comment
;; Scheme
(define addn
  (lambda (n)
    (let ((x n))
      (lambda (y) (+ x y)))))

;; Clojure
;; (defn name [args] body) macroexpands to (def name (fn [args] body))
;; fn is lambda

(defn addn [n]
  (let [x n]
    (fn [y] (+ x y))))
```

In addition, Clojure has reduced the nesting required in many constructs, like let:

```
;; Scheme
(let ((x 2) (y 3))
  (* x y))
```

```
;; Clojure
(let [x 2, y 3] ;; the comma is optional whitespace
  (* x y))
```

This combination of clarity around expressions and reduced nesting has been a great boost to Clojure's (subjective) 'readability' for beginners, without discarding code-as-data e.g., for C-like notation.

REFERENCES

Anglican. 2020. Anglican web site. https://probprog.github.io/anglican/ (also at Internet Archive 5 March 2020 15:36:41).

Phil Bagwell. 2001. Ideal Hash Trees. http://infoscience.epfl.ch/record/64398 Technical report, October 2001.

Philip Bagwell and Tiark Rompf. 2011. RRB-Trees: Efficient Immutable Vectors. 16. http://infoscience.epfl.ch/record/169879

Henry G. Baker. 1993. Equal rights for functional objects or, the more things change, the more they are the same. *ACM SIGPLAN OOPS Messenger* 4, 4 (Oct), 2–27. https://doi.org/10.1145/165593.165596

Philip A. Bernstein and Nathan Goodman. 1983. Multiversion Concurrency Control—Theory and Algorithms. *ACM Trans. Database Syst.* 8, 4 (Dec.), 465–483. https://doi.org/10.1145/319996.319998

Richard Bird et al. 1988. *Lectures on constructive functional programming*. Oxford University Computing Laboratory, Programming Research Group.

Linus Björnstam. 2019. SRFI-171 Transducers. https://srfi.schemers.org/srfi-171/srfi-171.html (also at Internet Archive 9 March 2020 17:40:11).

Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. 2016. Practical Optional Types for Clojure, Vol. 9632. (04), 68–94. https://doi.org/10.1007/978-3-662-49498-1_4

Cliff Click and John Rose. 2002. Fast Subtype Checking in the HotSpot JVM. In *Proceedings of the 2002 Joint ACM-ISCOPE Conference on Java Grande* (Seattle, Washington, USA) (*JGI '02*). Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/583810.583821

clojurebridge.org. 2020. ClojureBridge. https://clojurebridge.org/ (also at Internet Archive 21 Feb. 2020 12:24:36).

ClojureIRC. 2008. Clojure IRC log. http://clojure-log.n01se.net/date/2008-02-01.html (also at Internet Archive 10 March 2020 16:39:37).

clojure.org. 2019. Companies Using Clojure. https://clojure.org/community/companies (also at Internet Archive 5 March 2020 15:38:19).

clojure.org. 2020. Clojure Success Stories. https://clojure.org/community/success_stories (also at Internet Archive 5 March 2020 15:39:24).

ClojureTV. 2019. ClojureTV NA. https://www.youtube.com/user/ClojureTV
ClojureTV is a YouTube video channel, the description for which is:
"ClojureTV brings together talks and presentations from the community, major Clojure conferences, and Cognitect. Here you can find Clojure/conj, Clojure/west and EuroClojure talks, as well as custom video tutorials and presentations from Rich Hickey and the Cognitect team.".

Robert Crim and Jon Pither. 2016. Funding Circle – Lending some Clojure. https://juxt.pro/blog/posts/clojure-in-fundingcircle.html (also at Internet Archive 5 March 2020 15:51:29).

diab0lic. 2018. Clojure at Netflix. https://news.ycombinator.com/item?id=18346043 (also at Internet Archive 5 March 2020 15:44:58).

James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. 1989. Making data structures persistent. *Journal of computer and system sciences* 38, 1, 86–124.

edn. 2014. edn Spec. https://github.com/edn-format/edn (also at Internet Archive 5 March 2020 15:40:35).

edn. 2019. edn Implementations. https://github.com/edn-format/edn/wiki/Implementations (also at Internet Archive 24 Feb. 2020 20:57:00).

Michael Ernst, Craig Kaplan, and Craig Chambers. 1998. Predicate dispatching: A unified theory of dispatch. In *European Conference on Object-Oriented Programming*. Springer, 186–211.

Richard P Gabriel, Sonya E Keene, Gregor Kiczales, David A Moon, Jim Kempf, Larry Masinter, Mark Stefik, Daniel L Weinreb, and Jon L White. 1987. Common lisp object system specification.". *ANSI X3J13 Document*, 87–002.

Richard P Gabriel and Kent M Pitman. 1988. Endpaper: Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation* 1, 1, 81–101.

Igor Ges and Gerardo Castro. 2018. Clojure vs Sepsis: path to real time enterprise data science NA. https://www.youtube.com/watch?v=AyWbB52SzAg
This is a video recording of a talk given at the Clojure Conj 2018 conference, the abstract for which is:
"After being tasked with bringing data science to HCA, the largest hospital chain in the United States, we recognized

the need for a large scale, life-saving project to demonstrate value and pave the way for getting the infrastructure and support that a meaningful data science operation would require. This is the story of how we used Clojure and Datomic to successfully deploy SPOT (Sepsis Prevention and Optimization of Therapy) in 165 hospitals all across the US. In this talk we will discuss the design of a real-time streaming clinical system, our experience using Datomic for a medical data science application, as well as challenges that we had to overcome (both technical and organizational) to deploy and operate SPOT at this scale.".

Paul Graham. 2003. Lisp Essays. http://www.paulgraham.com/lisp.html (also at Internet Archive 1 Feb. 2020 09:49:40).

Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques.* Elsevier.

Stuart Halloway. 2009. *Programming Clojure.* Pragmatic Bookshelf.

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM, 48–60.

Rich Hickey. 1996. Callbacks in C++ using template functors. In *C++ gems.* SIGS Publications, Inc., 515–537.

Rich Hickey. 2003. DotLisp - A Lisp Dialect for .Net. http://dotlisp.sourceforge.net/dotlisp.htm (also at Internet Archive 10 Feb. 2020 20:30:16).

Rich Hickey. 2004. Jfli - A Java Foreign Language Interface for Common Lisp. http://jfli.sourceforge.net/ (also at Internet Archive 10 Feb. 2020 20:17:34).

Rich Hickey. 2011. Simple Made Easy NA. https://www.infoq.com/presentations/Simple-Made-Easy/
This is a video recording of a talk given at the Strange Loop 2011 conference, the abstract for which is:
"Simple has a core meaning, an understanding of which is critical to developing robust and flexible software. We should be simplifying the problem domain, and creating solutions by composing simple parts. Instead, we endure complexity, and pride ourselves in our ability to manage it. We can do better. This talk will discuss simplicity, why it is important, how to achieve it in your designs and how to recognize its absence in the tools, language constructs and libraries we use.".

Rich Hickey and Eric Thorsen. 2005. Foil - a Foreign Object Interface for Lisp. http://foil.sourceforge.net/ (also at Internet Archive 10 Feb. 2020 20:25:50).

Charles Antony Richard Hoare. 1978. Communicating sequential processes. In *The origin of concurrent programming.* Springer, 413–443.

IETF. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. https://tools.ietf.org/html/rfc8259 (also at Internet Archive 7 March 2020 20:01:42).

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) (*LFP '86*). Association for Computing Machinery, New York, NY, USA, 151–161. https://doi.org/10.1145/319838.319859

T Lindholm and F Yellin. 1999. The Java TM Virtual Machine Specification, 2nd edn. Sun Microsystems.

Anthony Marcar. 2015. Clojure at Scale NA. https://www.youtube.com/watch?v=av9Xi6CNqq4
This is a video recording of a talk given at the Clojure West 2015 conference, the abstract for which is:
"There are many resources to help you build Clojure applications. Most however use trivial examples that rarely span more than one project. What if you need to build a big clojure application comprising many projects? Over the three years that we've been using Clojure at WalmartLabs, we've had to figure this stuff out. In this session, I'll discuss some of the challenges we've faced scaling our team and code base as well as our experience using Clojure in the enterprise.".

Mark McGranaghan. 2011. Clojurescript: Functional programming for javascript platforms. *IEEE Internet Computing* 15, 6, 97–102.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. 1997. *The definition of standard ML: revised.* MIT press.

Ben Moseley and Peter Marks. 2006. Out of the tar pit. *Software Practice Advancement (SPA)* 2006.

Chris Okasaki. 1999. *Purely functional data structures.* Cambridge University Press.

Stack Overflow. 2019. Developer survey results 2019. https://insights.stackoverflow.com/survey/2019 (also at Internet Archive 5 March 2020 15:55:07).

Alan J Perlis. 1982. Special feature: Epigrams on programming. *ACM Sigplan Notices* 17, 9, 7–13.

Juan Pedro Bolívar Puente. 2015. Transducers: from Clojure to C++ NA. https://www.youtube.com/watch?v=vohGJjGxtJQ
This is a video recording of a talk given at CppCon 2015, the abstract for which is:
Transducers allow to us express transformations on sequential inputs (like std::transform, std::filter, most of boost::range::adaptors and more) in a way that is independent of the input source. They are simple high order functions agnostic of the notion of iterator or collection and can be combined by simple function composition. They can later be applied eagerly or lazily over iterators, but most interestingly, also to other kinds of "reactive" sources, like networked streams, inter-process channels or reactive observables (e.g. RxCpp). Not only they provide an elegant decoupling between the "what" and the "how" and higher level of reuse, their implementation is often simpler and and more performant than similar boost::range adaptors. Transducers were introduced in Clojure by Rich Hickey. At Ableton we implemented them in C++ and released them as part of our Open Source modern C++ toolkit: Atria. Our implementation

introduces innovations of its own, like implementing state-full transducers without mutation, enabling further safety and reusability. We also use Eric Niebler's technique to describe and check its concepts in standard compliant C++11. In this session we will introduce the concept of transducers and how they can be implemented in modern C++. We will also cover some of the most interesting use-cases.

Christopher Small. 2018. Clojure on the cyberpunk frontier of democracy NA. https://www.youtube.com/watch?v=2tBVMAm0-00

This is a video recording of a talk given at the Clojure Conj 2018 conference, the abstract for which is:
"Polis is an open source tool which uses data science to help groups build understanding and make decisions. At the core of this tool is a machine learning engine written in Clojure. As my very first Clojure project, and the reason I got into Clojure, this codebase offers a unique perspective from which to look back on my evolving use and understanding of the language. Along the way I hope to highlight the strengths of Clojure in data-science, and to explain what cybernetics actually is.".

source d. 2020. Hercules. https://github.com/src-d/hercules (also at Internet Archive 5 March 2020 15:42:27).

Guy L Steele. 1999. Growing a language. *Higher-Order and Symbolic Computation* 12, 3, 221–236.

Guy L Steele and Richard P Gabriel. 1996. The evolution of Lisp. In *History of programming languages—II*. ACM, 233–330.

Guy L Steele Jr, Scott E Fahlman, Richard P Gabriel, David A Moon, Daniel L Weinreb, Daniel G Bobrow, Linda G DeMichiel, Sonya E Keene, Gregor Kiczales, Crispin Perdue, et al. 1990. Waters, and Jon L White. Common Lisp: The Language.

Guy Lewis Steele Jr and Gerald Jay Sussman. 1978. *The Revised Report on SCHEME: A Dialect of LISP*. Technical Report. MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages* (Leuven, Belgium) (*IFL 2016*). Association for Computing Machinery, New York, NY, USA, Article Article 6, 12 pages. https://doi.org/10.1145/3064899.3064910

W3C. 2014. RDF 1.1 XML Syntax. https://www.w3.org/TR/rdf-syntax-grammar/ (also at Internet Archive 5 March 2020 13:23:43).

Philip Wadler and 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*. ACM, 60–76.

Edward Wible and Jeff Meyerson. 2018. Build a Bank: Nubank with Edward Wible. https://softwareengineeringdaily.com/2018/07/10/build-a-bank-nubank-with-edward-wible/ (also at Internet Archive 5 March 2020 15:45:47).