# Rich Hickey Q&A

by Michael Fogus

Best known as the inventor of Clojure, a Lisp that runs on the Java Virtual Machine and the first new member of the Lisp family to attract any widespread interest since Scheme and Common Lisp, Rich Hickey has been a software developer and consultant for two decades.

Prior to starting work on Clojure, he made four attempts to combine Lisp with either Java or Microsoft's Common Language Runtime: jfli, Foil, Lisplets, and DotLisp but Clojure was the first to draw significant attention. To date there have been four books published on Clojure, including *The Joy of Clojure* by interviewer Michael Fogus. The first Clojure conference, ClojureConj held in 2010, drew over two hundred attendees. And the Clojure Google group has, as of this writing, 4,880 members who have posted over 46,000 messages since it was started in 2007.

In past lives Hickey has been a music composition major, a guitarist, and a serious C++ programmer, even teaching an Advanced C++ course at New York University. In this interview, Michael Fogus talks with Hickey about complexity, abstraction, and the past, present, and future of Clojure.

Fogus: What drew you to programming?

Hickey: I was running a recording studio and got a computer for it. I was intrigued by what it could do—this was in the relatively early days of MIDI—and filled with the possibilities of what it might do. So I taught myself C and assembly language and started writing music software. I was immediately hooked.

Fogus: You play guitar, correct?

Hickey: Yes, I was a music composition major in college.

Fogus: I've often found that great programmers are skilled musicians. Do you think that musical art is related to the art of programming?

Hickey: I think the skills useful for one are often useful for the other. Coding and music performance are fairly different, other than being disciplines that reward practice, but software design and composition have a lot of similarities. Both involve manipulating and coordinating abstractions, and envisioning their realization, in the case of programs that are processes, in and over time. I've certainly found software design satisfies the creative urge I intended to pursue in composition, and has completely displaced it.

Fogus: What was the first interesting program that you wrote?

Hickey: An early program I wrote that got me excited about the possibilities of computing was a simple evolution program. Bugs moved around on the screen looking for food, with random movement to start. I would start it before bed, with the bugs shaking around, and awake to find they had evolved these fluid, efficient movement strategies. It was then that I realized that simulations and generative programs would allow us to see and discover things that analysis and calculation couldn't.

Fogus: What do you do to improve your skills as a programmer? Do you have certain exercises or pet projects that you use?

Hickey: I read relentlessly. I don't do any programming not directed at making the computer do something useful, so I don't do any exercises. I try to spend more time thinking about the problem than I do typing it in.

Fogus: Speaking of reading, you once created a Clojure Bookshelf list on Amazon that was very popular. Of those books listed, are there any you think every programmer should read?

Hickey: I couldn't highlight just a few. Each illuminates some things and ignores others. I'm not really comfortable advocating what others ought to do. Personally, I try to read, on an ongoing basis, books such as those on the list and papers from academia, many of which are available online. And I like to see a commitment to learning on the part of people with whom I work.

Fogus: Let's talk about some of your choices for the Clojure Bookshelf list. For example, on the surface Clojure seems to be very different from Ruby, yet you list *Programming Ruby 1.9* by Thomas, Fowler, and Hunt. How did that book, and Ruby in general, influence the design of Clojure?

Hickey: Well, influences can be positive or negative. Looking at Python and Ruby left me resolute that I didn't want to create yet another syntax and yet another object system. On the other hand, they set a high bar for concision.

Fogus: Do you think Ruby or Python has taken the ALGOL-derived syntax to the limit of its concision?

Hickey: I don't know. I'm more interested in reducing complexity than I am in concision.

Fogus: Let's explore that a little. There are the complexities of the problem, which are mostly language independent, and then there are incidental complexities imposed by the language itself. How does Clojure alleviate the last of these—the incidental complexities?

Hickey: Reducing incidental complexity is a primary focus of Clojure, and you could dig into how it does that in every area. For example, mutable state is an incidental complexity. The mechanics of it seem simple, or at least familiar, but the reality is quite complex. In my opinion, it is clearly the number one problem in systems. So, Clojure makes immutable data the default.

Since we were talking about syntax, let's look at classic Lisp. It seems to be the simplest of syntax, everything is a parenthesized list of symbols, numbers, and a few other things. What could be simpler? But in reality, it is not the simplest, since to achieve that uniformity, there has to be substantial overloading of the meaning of lists. They might be function calls, grouping constructs, or data literals, etc. And determining which requires using context, increasing the cognitive load when scanning code to assess its meaning. Clojure adds a couple more composite data literals to lists, and uses them for syntax. In doing so, it means that lists are almost always call-like things, and vectors are used for grouping, and maps have their own literals. Moving from one data structure to three reduces the cognitive load substantially.

As programmers we've become quite familiar with many incidental complexities, but that doesn't make them less complex, it just makes us more adept at overcoming them. But shouldn't we be doing something more useful?

Fogus: So once incidental complexities have been reduced, how can Clojure help solve the problem at hand? For example, the idealized object-oriented paradigm is meant to foster reuse, but Clojure is not classically object-oriented—how can we structure our code for reuse?

Hickey: I would argue about OO and reuse, but certainly, being able to reuse things makes the problem at hand simpler, as you are not reinventing wheels instead of building cars. And Clojure being on the JVM makes a lot of wheels—libraries—available. What makes a library reusable? It should do one or a few things well, be relatively self-sufficient, and make few demands on client code. None of that falls out of OO, and not all Java libraries meet this criteria, but many do.

When we drop down to the algorithm level, I think OO can seriously thwart reuse. In particular, the use of objects to represent simple informational data is almost criminal in its generation of per-piece-of-

information micro-languages, i.e. the class methods, versus far more powerful, declarative, and generic methods like relational algebra. Inventing a class with its own interface to hold a piece of information is like inventing a new language to write every short story. This is anti-reuse, and, I think, results in an explosion of code in typical OO applications. Clojure eschews this and instead advocates a simple associative model for information. With it, one can write algorithms that can be reused across information types.

This associative model is but one of several abstractions supplied with Clojure, and these are the true underpinnings of its approach to reuse: functions on abstractions. Having an open, and large, set of functions operate upon an open, and small, set of extensible abstractions is the key to algorithmic reuse and library interoperability. The vast majority of Clojure functions are defined in terms of these abstractions, and library authors design their input and output formats in terms of them as well, realizing tremendous interoperability between independently developed libraries. This is in stark contrast to the DOMs and other such things you see in OO. Of course, you can do similar abstraction in OO with interfaces, for instance, the `java.util` collections, but you can just as easily not, as in `java.io`.

Fogus: Can you expand on what you mean by “simple associative model for information”?

Hickey: Most classes used to represent information are just bespoke associative maps of named properties/attributes to values. But in the customization process we usually lose the ability to treat them like generic maps. This then precludes the writing of generic information manipulation code, since such code requires the capability to generically access/modify/add properties by name/key, enumerate properties, etc. An associative information model retains and emphasizes those capabilities.

Fogus: Are there any domains where this abstraction-oriented approach isn’t suitable?

Hickey: I think the approach in general has universal appeal. The pressure on abstractions, and on dynamic languages like Clojure, comes from the quest for performance. People seeking the utmost performance might not find Clojure suitable. They might not even find Java suitable, with its lack of composite value types both on the stack and in arrays.

That said, this is an area of ongoing improvement in Clojure, which is already pretty fast, as dynamic languages go. With optional type hints, Clojure can already generate local code as fast as Java, and new work on primitive arguments and returns is enabling that speed across larger scopes. Whether that can be carried to the highest levels of the abstraction stack, without incurring the rigidity and complexity of a full-on type system, is an open question I intend to pursue.

Fogus: In an old paper of yours, “Callbacks in C++ Using Template Functors”, you write favorably about C++, OOP, and static typing. Why did you change your mind?

Hickey: I’m not sure I did. I said C++ was flexible—it is—and that, when implementing a callback system for C++, one should remain aligned with its object orientation and static typing. More interesting to me, in rereading it, is that I am still now making the same arguments I made then, fifteen years ago, against mixins and derivation as extension mechanisms.

That said, I certainly was a fan of C++ in the day, and five more years of it cured me of that. The complexity is stunning. It failed as the library language it purported to be, due to lack of GC, in my opinion, and static typing failed to keep large OO systems from becoming wretched balls of mud. Large mutable object graphs are the sore point, and `const` is inadequate to address it. Once C++’s performance advantage eroded or became less important, you had to wonder—why bother? I can’t imagine working in a language without GC today, except in very special circumstances.

Along the way, I discovered Common Lisp, which was much more flexible, dynamic, simpler, and fast

enough, and decided that was how I wanted to program. Finally, with Clojure, that is becoming possible, and practical, for me.

Fogus: In an email exchange, you mentioned that during the process of learning Lisp, you experienced joy—a motivation for the title of my book, by the way. Can you elaborate on that feeling and why it seems that Lisp fosters such a feeling?

Hickey: You can reach a point with Lisp where, between the conceptual simplicity, the large libraries, and the customization of macros, you are able to write only code that matters. And, once there, you are able to achieve a very high degree of focus, such as you would when playing Go, or playing a musical instrument, or meditating. And then, as with those activities, there can be a feeling of elation that accompanies that mental state of focus.

Fogus: What programming languages have you used professionally?

Hickey: Mainly C, C++, Java, C#, Common Lisp, and Clojure.

Fogus: What is your second favorite programming language?

Hickey: If I had been more satisfied with any of those, I wouldn't have written Clojure. If I had to be stranded with something other than Clojure, I'd be happiest with a good Common Lisp and its source code. If I had more free time, I'd spend it with Haskell.

Fogus: I have a theory that the excitement surrounding Clojure is in part due to a general open-mindedness fostered by Paul Graham's original Lisp essays and the popularity of Python and Ruby. What do you attribute to Clojure's success thus far?

Hickey: I agree with your theory. I think Paul Graham's essays were hugely influential, and got people interested in Lisp, a Lisp-like way of thinking about programming, and the importance of rejecting conventional wisdom. And Python and Ruby—and PHP and Javascript—have helped herald a renaissance of language diversity, as people were obviously succeeding with languages other than Java/C#/C++. All of this paved the way for Clojure.

It's interesting, because Clojure provides almost nothing you can't find somewhere else. But I do think it occupies an otherwise empty spot in the multidimensional space of language features and capabilities. If it hadn't, I wouldn't have written it. That's the spot I wanted to work in, and enough other people must want to be there too.

Fogus: You've personally done a lot to set the tone in the Clojure community. How much does a language's community contribute to its success?

Hickey: I think it is a huge component. I am so happy with, and proud of, the Clojure community. People are helpful, and respectful, and positive. I think the key point is that the community values itself, such that people will decide it is more important to preserve the quality of the community than to vent their emotions or prove themselves right.

Fogus: Can you talk briefly about the Lisp-related projects leading up to the creation of Clojure? Specifically, what were the goals of dotLisp, Foil, and Lisplets?

Hickey: dotLisp was the inevitable rite of passage write-a-Lisp-interpreter thing. The only thing interesting about it was that, like Clojure, it was designed to be hosted and provide convenient access to the host, the CLR in this case.

Jfli was next, an attempt to provide access to Java by embedding a JVM inside a Common Lisp process. This worked okay, but still had a dissatisfying us-and-them feel.

Foil was essentially the same concept, but out of process. It used the same sexpr wire protocol to

support both Java and CLR interop. Still us-and-them, and slower than same process, but theoretically less intrusive.

Lisplets was even more decoupled, merely translating Java servlet requests and responses to sexprs so you could write your servlets in Lisp.

In the end, none of these really let you sneak Lisp into a more traditional shop, nor did they provide satisfyingly fast access to the abundant Java libs from Lisp.

Fogus: What lessons did you take away from those experiments when creating Clojure?

Hickey: That it was possible to create a satisfying Lispy syntax for accessing traditional OO stuff. That you really want to be on the same side of the fence, sharing GC, etc. with the host. The 'foreign' part of FFI has to go.

Fogus: Clojure was once in parallel development on both the JVM and the CLR, why did you eventually decide to focus in on the former?

Hickey: I got tired of doing everything twice, and wanted instead to do twice as much.

Fogus: Referring back to your previous comment regarding the negative aspect of influences, I'm led to wonder if the inclusion of *Prolog Programming for Artificial Intelligence* by Ivan Bratko to your Bookshelf was of this variety. You've mentioned elsewhere that the common view of Prolog as declarative is overblown—can I assume that Prolog negatively influenced Clojure?

Hickey: I didn't say overblown. I said it is less declarative than it might be, what with cut/fail and clause order dependence. On the other hand, it is much more declarative than what most of us are doing all the time, and serves as inspiration towards a more declarative approach. During the early development of Clojure, I built a prototype predicate dispatch system for it using a Lisp-based Prolog. It never became part of Clojure, but I am still interested in predicate dispatch, as well as using logic systems in place of a type system for Clojure. Definitely a positive influence, if somewhat under-delivered upon, as of yet.

Fogus: I have studied the Clojure Datalog implementation and am saddened that it does not get a lot of exposure. Do you think that there is a place for it, or some derivative, as the basis for that "logic system"?

Hickey: Yes, definitely. I like Datalog a lot.

Fogus: To what extent should a programming language be designed to prevent programmers from making mistakes or writing bad code?

Hickey: I'm reluctant to say "should", as different languages can rightly take different approaches to this. I know my personal focus is on enabling people to do the right thing rather than preventing them from doing the wrong thing. In the end, there is nothing that will prevent people from making mistakes or writing bad code.

Fogus: Following that idea—some people are surprised by the fact that Clojure does not engage in data-hiding encapsulation on its types. Why did you decide to forgo data-hiding?

Hickey: Let's be clear that Clojure strongly emphasizes programming to abstractions. At some point though, someone is going to need to have access to the data. And if you have a notion of "private", you need corresponding notions of privilege and trust. And that adds a whole ton of complexity and little value, creates rigidity in a system, and often forces things to live in places they shouldn't. This is in addition to the other losing that occurs when simple information is put into classes. To the extent the data is immutable, there is little harm that can come of providing access, other than that someone could come to depend upon something that might change. Well, okay, people do that all the time in real life,

and when things change, they adapt. And if they are rational, they know when they make a decision based upon something that can change that they might in the future need to adapt. So, it's a risk management decision, one I think programmers should be free to make.

If people don't have the sensibilities to desire to program to abstractions and to be wary of marrying implementation details, then they are never going to be good programmers.

Fogus: Where can we draw the line between sensibilities and language philosophy? That is, could the same be said for immutability in that we could simply say that programmers should follow a convention of immutability instead of it being enforced by the language?

Hickey: There's no such thing as a convention of immutability, as anyone who has tried to enforce one can attest. If a data structure offers only an immutable API, that is what's most important. If it offers a mixed API, it's simply not immutable.

Enforcement is orthogonal. That's not to say there isn't value in enforcement, as many optimizations can come into play. But there's no free lunch—type systems that can enforce purity are complex.

Fogus: What would you say to people who claim that Clojure is not a "real Lisp"?

Hickey: Life is too short to spend time on such people. Plenty of Lisp experts have recognized Clojure as a Lisp. I don't expect everyone to prefer Clojure over their favorite Lisp. If it wasn't different in some ways, there'd be little reason for it to exist.

Fogus: Aside from an obvious language choice like Lisp-1 vs. Lisp-2, how does Clojure differ from and hope to improve on Common Lisp and Scheme?

Hickey: The two most significant changes are: the core library is implemented in terms of abstractions, not concrete data types, e.g. sequence and associative abstractions rather than cons cells, and the core data structures are immutable and persistent.

Fogus: Referring back to your previous statement about Clojure allowing Lisp to be sneaked into traditional shops—how does Clojure differ in this respect from other JVM-based Lisps?

Hickey: Not much. You can sneak in almost any JVM language similarly.

Fogus: You've said you've been surprised by how popular Clojure has become, but on the other hand didn't you bet a couple years of your life with little or no other income to produce the first version?

Hickey: I started it while on a sabbatical I had given myself. Not a break from work, but a break *to* work, as a completely free person. I gave myself leave to do whatever I thought was right, with no regard for what others might think, nor any motivation to profit. In releasing it, I had the normal expectations for a new language—that ten to a hundred people might use it. Maybe I would get some help or code contributions.

It has taken off, and subsequently demanded far more time than the sabbatical I planned. So, I'm trying to recoup some of the investment I've made. Had it been a financially motivated undertaking, I'm sure Clojure would not exist, but I don't regret having invested in it.

Fogus: You released a series of videos introducing Clojure that generated serious buzz around the language. In my opinion they are a brilliant marketing strategy, especially for a young language. Were you intentionally creating those videos as marketing material, or was that simply a side effect of a purely informational pursuit?

Hickey: I've never intentionally marketed Clojure, other than the first email announcing its existence to the very few members of the jfli and Foil mailing lists.

I've given many invited talks, and the videos are recordings of some of those talks. It just seemed like a

sensible way to leverage the effort that went into doing the talks. I was quite surprised by the audience they received, but it proves that videos like that are much more efficient than talking to fifty to a hundred people at a time.

Fogus: As someone who only knows Haskell enough to read the papers, Clojure appears to be influenced by it substantially. From the names and operation of core functions—`take`, `drop`, `iterate`, `repeat`, etc.—to its protocols facility, there is a lot in Clojure that a Haskell programmer would recognize. Can you elaborate on Haskell's influences on Clojure both positive and negative?

Hickey: I think Haskell is a fantastic, awe-inspiring piece of work. I haven't used it in anger, but it certainly was a positive influence. Haskell obviously goes much further than Clojure in pursuing the ideals of functional programing. In particular they differ in the approach to using types to enforce things.

In some ways, Clojure is an experiment to see how many of the benefits of functional programming can be delivered without static enforcement. Certainly Clojure shows that you can get many benefits of using immutable data and pure functions merely by supplying them as defaults and choosing to use them, much in the same way you can get the benefits of walking on the sidewalk without there being guard rails forcing you to stay on the sidewalk.

I think the great challenge for type systems in practical use is getting them to be more expressive without a corresponding—or worse—increase in complexity. I have yet to see that, so they are not aligned with my desire to reduce complexity in programing.

As far as protocols go, they are as much akin to Common Lisp's generic functions as to Haskell's type classes, both of which demonstrate it is more flexible and extensible to keep functions and data separate, than to combine them as in typical OO.

Fogus: It's clear that protocols are influenced by CLOS. However, while CLOS allows you to build complex class hierarchies, Clojure's types and protocols do not. Can you comment on the problems associated with class hierarchies and how protocols address them?

Hickey: One way to think about inheritance and hierarchy is as a form of logical implication—if X is a Y, then all the things that are true of Y's are true of X's. The problems come about when you attach something to the hierarchy. If Y is just an interface, then it's relatively easy to make X satisfy it without conflict or contradiction. If Y is behavior and/or data, then things get dangerous quickly. There's more potential for conflict and contradiction, and, usually, there's also a method for partial overriding of the inheritance and thus, qualification of the isa implication. The implication is broken and your ability to reason about things turns to mud. And then of course there are the type-intrusion problems of inheritance-based designs.

Protocols and datatypes generally eschew implementation inheritance, and support interface inheritance for interop only. Protocols support direct connections of datatypes to protocols, without any inheritance. And protocols support direct implementation composition, which, in my opinion, is far preferable to inheritance for that purpose. You can still get implementation inheritance by extending protocols to interfaces, but that is a necessary compromise/evil for interop purposes, and should be used with care.

Fogus: Protocols and datatypes provide the basis for a bootstrapped Clojure—how important is it to implement Clojure in Clojure?

Hickey: It is important to be able to implement Clojure in Clojure, in order to make sure it has sufficient facilities to implement its data structures and algorithms. We are implementing any new data structures this way, and it is working out well. As far as going back and redoing things, I think the most

important bit is the Clojure compiler. It currently is a lot of Java, and no fun to maintain. In addition, there are several things I'd like to do differently with it in order to provide better support for tools. Next most important would be to move the abstractions from interfaces to protocols. Finally, a full bootstrap would make ports easier.

Fogus: Different target hosts would naturally support different subsets of Clojure's functionality. How do you plan to unify the ports?

Hickey: I don't. It has not been, and will not be, the objective of Clojure to allow porting of large programs from one host to another. That is simply a waste of time, and needed by almost no one. Currently, you often have to change languages when you change hosts—from Java to C# or Javascript. This is better than that, while short of some full portability layer. The idea is to be able to take one's knowledge of Clojure and its core libraries, and of the host du jour, and get something done. Certainly, non-IO libraries, like Clojure's core, can move between hosts. The JVM and CLR have rough capability parity. We'll have to see how restrictive a Javascript host might be.

Fogus: Will you formally define a "Clojure Kernel" as part of the Clojure-in-Clojure process?

Hickey: I doubt it. Perhaps after a few ports exist we can put a label on the commonality, but trying to do such formalization prior to getting a few hosts under your belt seems folly.

Fogus: Favorite tools? Editor? Version control? Debugger? Drawing tool? IDE?

Hickey: Circus Ponies NoteBook, OmniGraffle, hammock.[1]

Fogus: You have been known to speak out against test-driven development. Do you mind elaborating on your position?

Hickey: I never spoke out 'against' TDD. What I have said is, life is short and there are only a finite number of hours in a day. So, we have to make choices about how we spend our time. If we spend it writing tests, that is time we are not spending doing something else. Each of us needs to assess how best to spend our time in order to maximize our results, both in quantity and quality. If people think that spending fifty percent of their time writing tests maximizes their results—okay for them. I'm sure that's not true for me—I'd rather spend that time thinking about my problem. I'm certain that, for me, this produces better solutions, with fewer defects, than any other use of my time. A bad design with a complete test suite is still a bad design.

Fogus: Clojure provides function constraints via pre- and post-condition checks that provide a subset of Eiffel's contracts programming. Do constraints eliminate the need for, or complement unit testing?

Hickey: They complement unit tests. They have a number of nice properties—they document the intent of the code at the point it is written, and can optionally run in the context of the program.

Fogus: It seems that your decision to include features in Clojure is orthogonal to their implementation and inherent complexities. For example, it seemed that streams were right on the cusp of being integrated but were discarded outright. Likewise, scopes are relatively simple to comprehend and implement, but likewise seem to have been dropped, or at least delayed greatly. What are the reasons that you stepped away from these two in particular, and in general, what is your ultimate criteria for adding new features to Clojure?

Hickey: The default position is to not add features. Complexity does matter.

Streams, in particular, exposed some difficult things in the API with which I wasn't comfortable. Now some of the motivating ideas have moved into pods, where they make more holistic sense. Various features interact, e.g. pods, transients, and references, so you can't look at any one in isolation. Scopes may seem easy to implement, but only in ways that suffer the same limitations as vars and binding vis-

à-vis thread-pool threads. I have ideas about how to do that and binding better, and that work may have to precede delivering scopes. Scopes are still on the table.

I'd like for any new features to be actually needed and have designs I feel good about. It is a process that requires exploratory work, and time to think. I reserve the right to come up with a better idea, and sometimes I am just allocating time to do that by waiting. I like to think I don't primarily work on features—I work on problems that features help solve. Scopes are a feature but resource management is a problem; streams and pods are features but process is a problem. As you work on problems, you develop—and sometimes abandon—ideas for features.

Fogus: I've spoken with a few of your former co-workers, and they described you as a trouble-shooting and debugging master. How do you debug?

Hickey: I guess I use the scientific method. Analyze the situation given the available information, possibly gathering more facts. Formulate a hypothesis about what is wrong that fits the known facts. Find the smallest possible thing that could test the hypothesis. Try that. Often this will involve constructing an isolated reproducing case, if possible. If and only if the hypothesis is confirmed by the small test, look for that problem in the bigger application. If not, get more or better facts and come up with a different idea. I try to avoid attempting to solve the problem in the larger context, running in the debugger, just changing things to see effects, etc.

Ideally, you know you have solved the problem before you touch the computer, because you have a hypothesis that uniquely fits the facts.

Fogus: Is there a fundamental difference between debugging imperative/OO code versus Clojure code?

Hickey: There is no fundamental difference, but debugging functional code is much easier because of the better locality.

Fogus: Clojure's threaded concurrency story is very solid with numerous flavors of reference types providing different usage scenarios. Do you feel satisfied with Clojure's current concurrency offerings, or do you have plans to expand on the current reference model, or perhaps venture into distributed concurrency?

Hickey: Over time I've come to see this as more of a state/identity/value/time/process thing rather than concurrency in and of itself. Obviously it matters greatly for concurrent programs. I think there is room for at least one more reference type. To the extent one value is produced from another via a transient process, you could have a construct that allowed that process to have extent and/or multiple participants. This is the kind of thing people do on an ad hoc basis with locks, and could be wrapped in a reference-like construct, pods, that would, like the others, automate it, and make it explicit and safe.

I don't see distributed concurrency as a language thing. In addition, I don't think most applications are well served with directly connected distributed objects, but would be better off with some sort of message queues instead.

Fogus: While there are also primitives supporting parallelism, Clojure's story here has a lot of room for expansion. Do you plan to include higher-level parallel libraries such as those for fork-join or dataflow?

Hickey: Yes, there are plans, and some implementation work, to support fork-join–based parallel map/reduce/filter etc. on the existing data structures.

Fogus: Are high-level languages harder to optimize?

Hickey: I have no idea. What I do know is that, as we get to more virtualization, adaptive runtimes, dynamic compilation, etc., it is becoming more difficult to obtain a deterministic performance model

for all languages on such runtimes. This is presumably a trade-off to get better performance than we could obtain through manual optimization.

Fogus: You've cited the philosophy of Alfred North Whitehead—in particular his works *Process and Reality* and *Science and the Modern World*—in explaining Clojure's notion of state, time, and identity. What can we, as programmers, learn from Whitehead specifically and philosophy in general? Is there a place for philosophy in the education of software developers?

Hickey: I am not a proponent of the philosophy or metaphysics of Whitehead, and could hardly claim to understand it all. I was putting together a keynote for the JVM language summit and striving to find language-independent core ideas in the Clojure work. I was reminded of some Whitehead I had studied in college, so opened up a few of his books. Sure enough, he was all over some of the themes of my talk—time, process, immutability, etc. He is quite quotable, so I made him the 'hero' of the talk. But Whitehead was not an inspiration for Clojure—any connections were a serendipitous discovery after the fact. That said, the number of connections was startling.

To the extent we create simplified models of the world, like object systems, as programming constructs, yes, I guess any broader understanding of the world could benefit programmers.

1 Hickey has dubbed his previously mentioned tendency to spend more time thinking about problems than typing in code "Hammock-Driven Development".