Ghosts in the machine

Daniel Szmulewicz

October 7, 2016

Intro

Bio

- Ex-industry programmer, now running own business.
- A SAAS product in the field of e-commerce
- Author of system, a Clojure FLOSS library
- Founder of the Clojure user group in Israel
- Background in human sciences: philosophy, journalism, fiction writing.

system

`system` facilitates bottom-up programming for Clojure development. It tries to ensure that source code remains equivalent with the runtime state of the application under development. System achieves this goal by wrapping and leveraging three fundamental pieces: `tools.namespace`, `Component` and `Boot`. Introduces no semantics of its own.

Structure of talk

Interactive systems in general, particulars of Clojure's interactive story, live demo, and parting thoughts.

Domain of talk

- Programming techniques.
- Human–computer interaction.

- Epistemology / Psychology / Logic
- History of ideas in CS / Philosophy of CS.

MIT course: The Nature of Constructionist Learning (Media Arts and Sciences)

Definitions

Interactive programming

The procedure of writing parts of a program while it is already active.

Bottom up programming

Bottom-up programming is the opposite of top-down programming. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, gradually adding features until all of the application has been written.

Top-down programming

It refers to a style of programming where an application is constructed starting with a high-level description of what it is supposed to do, and breaking the specification down into simpler and simpler pieces, until a level has been reached that corresponds to the primitives of the programming language to be used.

Live coding

Live coding is a performing arts form and a creativity technique centred upon the writing of source code and the use of interactive programming in an improvised way. (The one that never works in conference talks.)

Interactive systems

A spiritual father

`https://media1.britannica.com/eb-media/13/19513-004-AFDA1514.jpg`
Piaget (1936) was the first psychologist to make a systematic study of cognitive development. His contributions include a theory of child cognitive development, detailed observational studies of cognition in children, and a series

of simple but ingenious tests to reveal different cognitive abilities. Piaget also had a considerable effect in the field of computer science and artificial intelligence. Seymour Papert used Piaget's work while developing the Logo programming language. Alan Kay used Piaget's theories as the basis for the Dynabook programming system concept, which was first discussed within the confines of the Xerox Palo Alto Research Center (Xerox PARC). These discussions led to the development of the Alto prototype, which explored for the first time all the elements of the graphical user interface (GUI), and influenced the creation of user interfaces in the 1980s and beyond.

> Marvin Minsky and Seymour Papert formed many of our attitudes about programming and its place in our intellectual lives. To them we owe the understanding that computation provides a means of expression for exploring ideas that would otherwise be too complex to deal with precisely. They emphasize that a student's ability to write and modify programs provides a powerful medium in which exploring becomes a natural activity.—SICP, Acknowledgments, Harold Abelson and Gerald Jay Sussman with Julie Sussman

Programming languages have been created, wholly or in part, for educational use, to support the constructionist approach to learning. Smalltalk was created as the language to underpin the "new world" of computing exemplified by "human–computer symbiosis." It was designed and created in part for educational use, more so for constructionist learning, at the Learning Research Group (LRG) of Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace, and others during the 1970s. The powerful built-in debugging and object inspection tools that came with Smalltalk environments set the standard for all the Integrated Development Environments, starting with Lisp Machine environments, that came after.

Non file-based environments

- APL (workspaces)
- Forth (blocks)
- Smalltalk (images)

File-based environments

Most mainstream languages, including statically typed languages, come with a REPL.

Lisp languages

Consult a paper (from 1978): Programming in an Interactive Environment: the "Lisp" Experience by Erik Sandewall

- Bootstrapping
- Incrementality
- Procedure orientation
- Internal representation of programs
- Full checking capability
- Declaration-free kernel
- Data structures and database
- Defined I/O for data structure
- Handles and interactive control

The kernel of the programming system must contain the following programs:

- a parser
- a program-printer
- an interpreter and/or
- a compiler

```
(loop (print (eval (read))))
```

> From within the environment provided by the REPL, you can define and redefine program elements such as variables, functions, classes, and methods; evaluate any Lisp expression; load files containing Lisp source code or compiled code; compile whole files or individual functions; enter the debugger; step through code; and inspect the state of individual Lisp objects.—Peter Seibel

Functionality of a Lisp REPL

- History of inputs and outputs.
- Variables for last result, last error (*1, *e).
- Help and documentation for commands. (`doc`, `source` in clojure.repl namespace)
- Variables to control the reader. (`*data-readers*`, `*default-data-reader-fn*`)
- Variables to control the printer. (`*print-length*`, `*print-level*`)

A REPL was never enough

> But for the true Lisp programming experience, you need an environment, such as SLIME, that lets you interact with Lisp both via the REPL and while editing source files. — **Peter Seibel**, Practical Common Lisp

> For instance, you don't want to have to cut and paste a function definition from a source file to the REPL or have to load a whole file just because you changed one function; your Lisp environment should let us evaluate or compile both individual expressions and whole files directly from your editor. — **Peter Seibel**, Practical Common Lisp

> Interlisp-D was notable for the integration of interactive development tools into the environment, such as a debugger, an automatic correction tool for simple errors (DWIM – "do what I mean"), and analysis tools.

Lisp systems

IDE	Non-IDE
Interlisp-D	Common Lisp
Racket	Scheme
Allegro	Clojure
Lispworks	Clojurescript

Special purpose environments for non-IDE Lisp systems in Emacs: Slime, Geiser.

Clojure

- Typical Lisp workflow: incremental compilation until it breaks. Restart/Respawn a process with new REPL.
- Clojure: Restarts are costly.

Clojure needs to bootstrap itself inside the JVM each and every time. Restart the application instead. Reloading the namespaces. Cojure has built-in facilities: `(require ... :reload)` and `(require ... :reload-all)` Pitfalls ahead because many Clojure abstractions were not designed with interactive mode in mind.

- If you modify two namespaces which depend on each other, you must remember to reload them in the correct order to avoid compilation errors.
- If you remove definitions from a source file and then reload it, those definitions are still available in memory. If other code depends on those definitions, it will continue to work but will break the next time you restart the JVM.
- If the reloaded namespace contains defmulti, you must also reload all of the associated defmethod expressions.
- If the reloaded namespace contains defprotocol, you must also reload any records or types implementing that protocol and replace any existing instances of those records/types with new instances.
- If the reloaded namespace contains macros, you must also reload any namespaces which use those macros.
- If the running program contains functions which close over values in the reloaded namespace, those closed-over values are not updated. (This is common in web applications which construct the "handler stack" as a composition of functions.)

Live coding part 1: old definitions, macros, protocols.

tools.namespace

Solved by tools.namespace single API call `refresh`. A Clojure contrib library. The `refresh` function will scan all the directories on the classpath for

Clojure source files, read their ns declarations, build a graph of their dependencies, and load them in dependency order. (You can change the directories it scans with set-refresh-dirs.) But first, it will wipe those namespace to clear out any old definitions. `remove-ns`. `remove-ns` doesn't dereference objects in memory, it just undoes the mapping between a namespace and its Vars. `remove-ns` doesn't "unload" code in any significant way.

Memory leak

A heap dump analysis of a reloaded application will reveal a memory leak (increase in allocated space per Var) of order O(n).

Threads

If a thread references a Var that gets unmapped, that thread will never see the Var's new value. The promise of Vars as stable references withers away.

Live coding part 2

Ghosts in the machine

Solution

Calling `remove-ns` should result in a garbage collection pass.

Takeaways

- Semantics of Clojure are explained with the implicit assumption of file-based development. Not interactively.
- Some abstractions leak in interactive mode.
- Clojure's interactive story is left as a tooling exercise for the community.
- Clojure the language comes with a lacking implementation for namespace removal, and makes no excuse for it aside from "Here be dragons".
- A REPL alone is never enough. Lots to catch up with systems from the sixties and seventies.