Clojure

Values and Change - Clojure's approach to Identity and State.

Many people come to Clojure from an imperative language and find themselves out of their element when faced with Clojure's approach to doing things, while others are coming from a more functional background and assume that once they leave Clojure's functional subset, they will be faced with the same story re: state as is found in Java. This essay intends to illuminate Clojure's approach to the problems faced by imperative and functional programs in modeling the world.

Imperative programming

An imperative program manipulates its world (e.g. memory) directly. It is founded on a now-unsustainable single-threaded premise - that the world is stopped while you look at or change it. You say "do this" and it happens, "change that" and it changes. Imperative programming languages are oriented around saying do this/do that, and changing memory locations.

This was never a great idea, even before multithreading. Add concurrency and you have a real problem, because "the world is stopped" premise is simply no longer true, and restoring that illusion is extremely difficult and error-prone. Multiple participants, each of which acts as though they were omnipotent, must somehow avoid destroying the presumptions and effects of the others. This requires mutexes and locks, to cordon off areas for each participant to manipulate, and a lot of overhead to propagate changes to shared memory so they are seen by other cores. It doesn't work very well.

Functional programming

Functional programming takes a more mathematical view of the world, and sees programs as functions that take certain values and produce others. Functional programs eschew the external 'effects' of imperative programs, and thus become easier to understand, reason about, and test, since the activity of functions is completely local. To the extent a portion of a program is purely functional, concurrency is a non-issue, as there is simply no change to coordinate.

Working Models and Identity

While some programs are merely large functions, e.g. compilers or theorem provers, many others are not - they are more like working models, and as such need to support what I'll refer to in this discussion as ***identity***. By identity I mean ***a stable logical entity associated with a series of different values over time***. Models need identity for the same reasons humans need identity - to represent the world. How could it work if identities like 'today' or 'America' had to represent a single constant value for all time? Note that by identities I don't mean names (I call my mother Mom, but you wouldn't).

So, for this discussion, an identity is an entity that has a state, which is its value at a point in time. And ***a value is something that doesn't change***. 42 doesn't change. June 29th 2008 doesn't change. Points don't move, dates don't change, no matter what some bad class libraries may cause you to believe. Even aggregates are values. The set of my favorite foods doesn't change, i.e. if I prefer different foods in the future, that will be a different set.

Identities are mental tools we use to superimpose continuity on a world which is constantly, functionally, creating new values of itself.

Object Oriented programming (OO)

OO is, among other things, an attempt to provide tools for modeling identity and state in programs (as well as associating behavior with state, and hierarchical classification, both ignored here). OO typically unifies identity and state, i.e. an object (identity) is a pointer to the memory that contains the value of its state. There is no way to obtain the state independent of the identity other than copying it. There is no way to observe a stable state (even to copy it) without blocking others from changing it. There is no way to associate the identity's state with a different value other than in-place memory mutation. In other words, ***typical OO has imperative programming baked into it!*** OO doesn't have to be this way, but, usually, it is (Java/C++/Python/Ruby etc).

People accustomed to OO conceive of their programs as mutating the values of objects. They understand the true notion of a value, say, 42, as something that would never change, but usually don't extend that notion of value to their object's state. That is a failure of their programming language. These languages use the same constructs for modeling values as they do for identities, objects, and default to mutability, causing all but the most disciplined programmers to create many more identities than they should, creating identities out of things that should be values etc.

Clojure programming

There is another way, and that is to separate identity and state (once again, indirection saves the day in programming). We need to move away from a notion of state as "the content of this memory block" to one of "the ***value*** currently associated with this identity". Thus an identity can be in different states at different times, but *the state itself doesn't change*. That is, an identity is not a state, an identity ***has*** a state. Exactly one state at any point in time. And that state is a true value, i.e. it never changes. If an identity appears to change, it is because it becomes associated with different state values over time. This is the Clojure model.

In Clojure's model, value calculation is purely functional. Values never change. New values are functions of old, not mutations. But logical identity is well supported, via atomic references to values (Refs and Agents). Changes to references are controlled/coordinated by the system - i.e. cooperation is not optional and not manual. The world moves forward due to the cooperative efforts of its participants and the programming language/system, Clojure, is in charge of world consistency management. The value of a reference (state of an identity) is always observable without coordination, and freely shareable between threads.

It is worth constructing programs this way even when there is only one participant (thread). Programs are easier to understand/test when functional value calculation is independent of identity/value association. And it's easy to add other participants when they are (inevitably) needed.

Concurrency

Dealing with concurrency means giving up the illusion of omnipotence. A program must recognize there will be other participants, and the world will keep changing. So a program must understand that if it observes the values of the states of some identities, the best it can get is a snapshot, as they can subsequently acquire new states. But often that is good enough for decision making or reporting purposes. We humans do quite well with the snapshots provided by our sensory systems. The nice thing is any such state value won't change in hand during the processing, as it is immutable.

On the other hand, changing state to a new value requires access to the 'current' value and the identity. Clojure's Refs and Agents handle this automatically. In the case of Refs, any interaction you do must occur within a transaction (else Clojure will throw an exception), all such interaction will see a consistent view of the world as of a point in time, and no changes will proceed unless the states to be changed haven't been changed by other participants in the meantime. Transactions support synchronous change to multiple Refs. Agents, OTOH, offer asynchronous change to a single reference. You pass a function and values, and, at some point in the future, that function will be passed the current state of the Agent and the return value of the function will become the Agent's new state.

In all cases the program will see stable views of the values in the world, as those values can't change, and sharing them among cores is fine. The trick is, "values never change" means that making new values from old ones must be efficient, and it is in Clojure, due to its persistent data structures. They allow you to finally follow the oft-proffered advice to favor immutability. So you set the state of an identity to a new state by reading its current value, calling a pure function on that value to create a new value, and setting that value as the new state. These composite operations are made easy and atomic by the alter, commute and send functions.

Message Passing and Actors

There are other ways to model identity and state, one of the more popular of which is the message-passing actor model, best exemplified by the quite impressive Erlang. In an actor model, state is encapsulated in an actor (identity) and can only be affected/seen via the passing of messages (values). In an asynchronous system like Erlang's, reading some aspect of an actor's state requires sending a request message, waiting for a response, and the actor sending a response. It is important to understand that *the actor model was designed to address the problems of **distributed** programs*. And the problems of distributed programs are much harder - there are multiple worlds (address spaces), direct observation is not possible, interaction occurs over possibly unreliable channels, etc. The actor model supports transparent distribution. If you write all of your code this way, you are not bound to the actual location of the other actors, allowing a system to be spread over multiple processes/machines without changing the code.

I chose not to use the Erlang-style actor model for same-process state management in Clojure for several reasons:

- It is a much more complex programming model, requiring 2-message conversations for the simplest data reads, and forcing the use of blocking message receives, which introduce the potential for deadlock. Programming for the failure modes of distribution means utilizing timeouts etc. It causes a bifurcation of the program protocols, some of which are represented by functions and others by the values of messages.
- It doesn't let you fully leverage the efficiencies of being in the same process. It is quite possible to efficiently directly share a large immutable data structure between threads, but the actor model forces intervening conversations and, potentially,

copying. Reads and writes get serialized and block each other, etc.

- It reduces your flexibility in modeling - this is a world in which everyone sits in a windowless room and communicates only by mail. Programs are decomposed as piles of blocking switch statements. You can only handle messages you anticipated receiving. Coordinating activities involving multiple actors is very difficult. You can't observe anything without its cooperation/coordination - making ad-hoc reporting or analysis impossible, instead forcing every actor to participate in each protocol.
- It is often the case that taking something that works well locally and transparently distributing it doesn't work out - the conversation granularity is too chatty or the message payloads are too large or the failure modes change the optimal work partitioning, i.e. transparent distribution isn't transparent and the code has to change anyway.

Clojure may eventually support the actor model for distributed programming, paying the price only when distribution is required, but I think it is quite cumbersome for same-process programming. YMMV of course.

Summary

Clojure is a functional language that explicitly supports programs as models and provides robust and easy-to-use facilities for managing identity and state in a single process in the face of concurrency.

In coming to Clojure from an OO language, you can use one of its persistent collections, e.g. maps, instead of objects. Use values as much as possible. And for those cases where your objects are truly modeling identities (far fewer cases than you might realize until you start thinking about it this way), you can use a Ref or Agent with e.g. a map as its state in order to model an identity with changing state. If you want to encapsulate or abstract away the details of your values, a good idea if they are non-trivial, write a set of functions for viewing and manipulating them. If you want polymorphism, use Clojure's multimethods.

In the local case, since Clojure does not have mutable local variables, instead of building up values in a mutating loop, you can instead do it functionally with recur or reduce.