

Re: Why I don't believe in static typing

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2003-11/1883.html>

From: Joachim Durchholz (joachim.durchholz_at_web.de)

Date: 11/18/03

Date: Tue, 18 Nov 2003 17:52:49 +0100

Isaac Gouy wrote:

- > *Maybe the interesting question is "How can we program systems which*
- > *behave in a reasonable manner in the presence of software errors?"*
- >
- > *The Erlang community seems to have some coherent ideas about that*
- > *question.*

For those who don't want to dig it up, here's the Erlang "philosophy" in a nutshell, filtered through my understanding:

- 1) Decouple the task at hand into many (>> 100, often >> 10.000) processes. (You need your own scheduler to do that, OS schedulers usually aren't up to that task. Processes also need to be extremely lightweight, both in terms of size overhead and in terms of process switching time.)
- 2) Use copying semantics to communicate between processes. No "shared memory", at least not conceptually. That way, if one process crashes (something that we expect since software is buggy), it will not be able to affect the data that other processes are working with. (This is what Erlangers mean if they say "message passing semantics" – the messages are copied, not transmitted by reference.)
- 3) If a process runs into a problem, let it crash as quickly as possible. "Problem" in this context means /uncorrectable/ problem, that is, if the programmer has no idea what the process should do in some specific circumstance. (Actually the advice goes even further: if the specification is incomplete and a situation occurs that's not described, the programmer shouldn't try to fill in the gaps, he should simply throw an exception that will crash the process.)
- 4) Processes have supervisor processes. If a process crashes, the supervisor is reliably informed that the process has crashed. Usually, the supervisor should also get a stack dump and other information that might help diagnose the crash, but that's more for giving the programmers a clue than for the operation of the system itself. (It's still important, of course.)
- 5) If a supervisor finds that a child has crashed, it should restart it (possibly after restarting other processes that must be restarted together with it). This is to work around transient problems, such as read errors, nonfatal race conditions (they tend to be nonfatal because

processes crash at the earliest opportunity, i.e. usually before any corrupted data makes it into a permanent data pool).

6) If restarting results in an immediate crash (for some application-dependent definition of "immediate"), assume the failure is permanent. Either let the supervisor itself crash (causing a higher-level supervisor to inspect the problem). Alternatively, start a simpler (set of) process(es) that will render a restricted set of services. (In other words, if a telephone finds that its dialling process crashes permanently, shut it down. Let the process for receiving phone calls continue, or – if dialling and receiving processes are interrelated – shut down the receiving process and restart a simpler receiver process that doesn't interact with dialling.)

I haven't programmed with that paradigm, but from my experience with other paradigms, I'd say that this has potential.

It isn't entirely necessary to map this all to processes. Processes with a copying communication semantics are just the easiest and historically most reliable strategy for achieving fail-safe independence between program parts. A language that uses immutable data almost exclusively, offers strong barriers between those domains that do use mutable data, and has an Eiffel-style exception mechanism (designed for shutting down faulty software, not for shortcut control transfers) should be able to apply this mindset just as well.

Generally, it's a "let it crash, have other software layers correct the problem if possible, have these other layers diagnose it anyway" philosophy.

As a side benefit, it allows "aggressive programming". You don't litter your code with error handling, you simply trigger an exception if things go wrong unexpectedly.

Just my understanding, as far as I gleaned it from Joe Armstrong's doctoral thesis. (Thanks to Joe for explaining the issues so clearly, it made a lot of things explicit that would have remained implicit for me if not written down in his thesis.)

Regards,
Jo