DPI

De Programmatica *Ipsum*

DE PROGRAMMATICA IPSUM

Issue 071: Go

August 5th, 2024

Table of Contents

Issue 071: Go 5
The Age Of Concurrency 9
Rob Pike 27
Sir Tony Hoare 35

Issue 071: Go

August 5th, 2024

Welcome to the 71st issue of *De Programmatica Ipsum*, about *The Go Programming Language*

In this edition:

- We provide context to the meteoric ascension of Go[1] in the programming world.
- In the Library section[2], we review the work and legacy of Sir C. A. R. “Tony” Hoare[3].
- In our Vidéothèque section[4], we watch four videos by Rob Pike[5].

We would like to thank our patrons who generously contribute every month (or have contributed in the past) to our work and help us run this magazine.

Thank you so much! In alphabetical order: Adam Guest, Adrian Tineo Cabello, Benjamin Sheldon, Christopher Nascone, Colin Powell, Franz Lucien Moersdorf, Guillermo Ramos Álvarez, Jean-Paul de Vooght, Dr. Juande Santander-Vela, Patryk Matuszewski, Paul Hudson, Quico Moya, Roger Turner, and Szymon Licau.

Enjoy this issue! Please subscribe to our free newsletter[6] to stay updated about new releases, share the articles on social media, or contribute[7] if you would like to support our work with a donation via Liberapay[8].

Cover photo by Richard Lee[9] on Unsplash[10].

References

[1] https://deprogrammaticaipsum.com/the-age-of-concurrency/
[2] https://deprogrammaticaipsum.com/category/library/
[3] https://deprogrammaticaipsum.com/sir-tony-hoare/
[4] https://deprogrammaticaipsum.com/category/videotheque/
[5] https://deprogrammaticaipsum.com/rob-pike/
[6] https://deprogrammaticaipsum.com/newsletter/
[7] https://deprogrammaticaipsum.com/contribute/
[8] https://liberapay.com/
[9] https://unsplash.com/@brock222?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash
[10] https://unsplash.com/photos/two-white-swans-flying-in-the-air-HZvKrcQflA0?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

The Age Of Concurrency

By Adrian Kosmaczewski

Younger generations of software developers, including those who started their career during the *Bonanza* of zero-interest money and pre-pandemic times of the 2010s, might be sadly oblivious of one of the major changes in the programming world of the latest 20 years: the transition from single-core CPUs to multicore ones.

The change happened slowly at first, and then all of a sudden. Consumers started noticing that the "MHz" and "GHz" of the CPUs in their newer personal com-

puters stopped growing around 2000. To give you an idea, the first PC I bought in 1992 featured a 16 MHz 80386[1] CPU; in 2004, my new Power Mac G5[2] had a whooping 2.7 GHz PowerPC G5 chip inside. That means, an increase of 167 times in 12 years.

Fast-forward 20 years later, and I am writing this article on a Lenovo ThinkPad Carbon X1 with a 16-core 12th Gen Intel Core i7-1270P clocking at 4.8 GHz when in good mood, but usually hovering around 3.5, or maybe even a bit less. That is an increase of 1.8 times, at best… in 20 years.

(I know, I know. This comparison is not correct. Good luck explaining why is that to your non-techie friends.)

The Times They Are A-Changing

Herb Sutter[3], who unlike your non-techie friends knows a thing or two about programming, also noticed the trend and wrote a seminal article in 2002: "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software"[4], followed up by "The Trouble with Locks" (of which page 1[5] and page 2[6] are thankfully available on the Internet Archive), all published on (of course!) Dr. Dobb's Journal, a venerable magazine we talked about last month[7].

> *Concurrency is the next major revolution in how we write software. Different experts still have different opinions on whether it will be bigger than OO, but that kind of conversation is best left to pundits. For technologists, the interesting thing is that concurrency is of the same order as OO both in the (expected) scale of the revolution and in the complexity and learning curve of the technology.*

TL;DR: silicon circuits were quickly reaching the size of atoms, which meant that various quantum-level effects effectively prevented CPU engineers from squeezing more circuits in the same wafer. This unprecedented situation was the first major roadblock in the otherwise unstoppable Moore's Law[8], and consumers noticed.

The "Free Lunch" in the title referred to the fact that, to make software run faster, developers from the seventies to the nineties just had to wait for a faster CPU, which was bound to come every 18 months in average, Pentium FDIV bugs[9]

notwithstanding. If CPUs could not become substantially faster, then programmers would have, for the first time in decades, to actually write efficient algorithms.

I know. The ignominy. But, hey, TANSTAAFL[10].

Multicore To The Rescue

The solution proposed by the hardware industry was quite simple, really: just bundle many CPUs into one, global warming be damned. This is how in 2006 I ended up buying a brand-new MacBook with a CPU featuring the brand new Intel Core architecture[11], an x86-compatible chip hosting two or more individual CPUs working in “parallel”. Or so said Apple’s marketing material.

Was this computer faster than its predecessors? Only marginally. The truth is that most software of that era seldom used that “parallel” capacity at full extent. Because no, most operating systems and most programming languages were not equipped in 2006 to enjoy the raw power of such multicore architectures.

The idea of a program being able to execute various tasks simultaneously, however, is quite old. Some readers are expecting more canonical sources for this claim, but stay with me: in the 1983 superhero movie “Superman III”[12], Gus Gorman (interpreted by the legendary Richard Pryor) “intuitively” learns how to write a concurrent program[13] calculating “two bilaterial coordinates at the same time”, whatever that is. We can see his work come to life on an early computer terminal showing two series of scrolling text, one after the other. Interesting fact: Gus wrote this program in BASIC[14] (of course!) and, if you look closely at the screen, with plenty of `PRINT` statements, as one does.

But let us go back to actual programming superheroes. Herb Sutter being a C++ superhero, let us check with him, was C++ ready to embrace concurrency in 2006? As Scott Meyers[15] (another C++ superhero) said in 2005, the answer is an outstanding no.

> *As a language, C++ has no notion of threads – no notion of concurrency of any kind, in fact. Ditto for C++’s standard library. As far as C++ is concerned, multithreaded programs don’t exist.*

("Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs" by Scott Meyers, 2005, page 9.)

Ouch. Indeed, C++ developers would have to wait until C++11[16] was approved by the committee to get any kind of concurrency support built-in on the standard. What C++ programmers got was quite transformational, including a new memory model and the support for lambda functions (more on that soon), and a few more interesting libraries:

> *The important point about `future` and `promise` is that they enable a transfer of a value between two tasks without explicit use of a lock; "the system" implements the transfer efficiently.*

("The C++ Programming Language, Fourth Edition", Bjarne Stroustrup, 2013, page 120.)

Prior Art

Concurrency poses some simple yet elusive problems:

> *The common wisdom is that the answer depends on the task in question: if a single person can dig at a rate of one cubic meter per hour, then in one hour a hundred people can dig a ditch that is 100 m long, but not a hole 100 m deep. Determining which computational tasks can be "parallelized" when many processors are available and which are "inherently sequential" is a basic question for both practical and theoretical reasons.*

("Computational Complexity", by Oded Goldreich and Avi Wigderson, section 5.1.3 of chapter IV.20, page 587, in "The Princeton Companion to Mathematics", Princeton University Press, 2008.)

Intuitively, concurrency appears as a good candidate for optimizing most computationally intensive tasks, like sorting and searching. Precisely, in 1998, Addison-Wesley published the first boxed set of "The Art of Computer Programming"[17] by Donald Knuth. In page 389 of the third volume, "Sorting and Searching", we learn that

> *For example, the present world record for* terabyte sorting *– 1010 records of 100 characters each – is 2.5 hours, achieved in September 1997 on a Silicon Graphics Origin2000 system with 32 processors, 8 gigabytes of internal memory, and 559 disks of 4 gigabytes each. This record was set by a commercially available sorting routine called Nsort™, developed by C. Nyberg, C. Koester, and J. Gray using methods that have not yet been published.*

The Sort Benchmark Home Page[18] provides more results of the Nsort algorithm in more modern hardware. Continuing with Dr. Knuth, on page 286 of the second volume, "Seminumerical Algorithms", we read an interesting prophecy of his:

> *Perhaps highly parallel computers will someday make simultaneous operations commonplace, so that modular arithmetic will be of significant importance in "real-time" calculations when a quick answer to a single problem requiring high precision is needed.*

Predictions are risky business. The late Barry Boehm[19] himself, who passed away[20] in 2022 at 87 years old, did not get the 2020s right:

> *Assuming that Moore's Law holds, another 20 years of doubling computing element performance every 18 months will lead to a performance improvement factor of 220/1.5 = 213.33 = 10,000 by 2025. Similar factors will apply to the size and power consumption of the competing elements.*

("A View of the 20th and 21st Century Software Engineering", in Chapter 8 of "Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research", by Barry Boehm, Wiley, 2007, page 720.)

Sorry Barry, it did not hold. We are almost in 2025, and the only part you got right was the power consumption bit. Sadly.

The early solutions to concurrent programming were plenty, and they all sucked: threads, locks, semaphores, shared memory. In particular threads, the canonical answer to concurrency, were notoriously hard for developers to grasp, and in the opinion of John Ousterhout, they were a bad idea for most purposes[21].

Hence, early on many researchers tried to find solutions to the problem of dividing a program in many parallel chunks, and to have some kind of runtime taking care of the minutiae. Some illustrious examples are Cilk[22] and OpenMP[23], both implementing ideas from the "Dynamic Multithreading" programming model exposed by Cormen, Leiserson, Rivest, and Stein in their classic book "Introduction to Algorithms". Page 773 of the third edition (2009) states:

> *Dynamic multithreading allows programmers to specify parallelism in applications without worrying about communication protocols, load balancing, and other vagaries of static-thread programming. (…) Nested parallelism allows a subroutine to be "spawned," allowing the caller to proceed while the spawned subroutine is computing its result.*

Finally, we cannot obviate a formal pattern of concurrent interaction called "Communicating Sequential Processes"[24] or CSP, created by C.A.R "Tony" Hoare[25] and which had a dramatic impact in the development of concurrent programming languages:

> *However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor.*

("Communicating Sequential Processes"[26], C.A.R. Hoare, Communications of the ACM, Volume 21, Issue 8, pages 666–677, doi:10.1145/359576.359585, 1978.)

Functional Programming To The Rescue

But research was not advancing fast enough, and the Free Lunch was almost over. Google the search engine was born in 1998, and it required such massive amounts of compute power that, to become Google the company as we know it today, it started using concurrent programming models when the rest of the industry was not paying attention to them – or to Herb Sutter, for that matter.

One person who did pay attention to what was going on was Joel Spolsky[27], in a quote from 2005 we have used a few[28] times[29] already in this magazine, but which fits this month's article too well:

> *The very fact that Google invented MapReduce, and Microsoft didn't, says something about why Microsoft is still playing catch up trying to get basic search features to work, while Google has moved on to the next problem: building Skynet^{H}H^{H}H^{H}H the world's largest massively parallel supercomputer. I don't think Microsoft completely understands just how far behind they are on that wave.*

The most important people in programming understood that the carpet was being pulled from beneath the developers' feet. In page 314 of Biancuzzi and Warden's "Masterminds of Programming" (2009), Anders Hejlsberg[30] mentions concurrency as one of the most important challenges in the development of C#:

> *A lot of people have harbored hope that one could have a slash parallel switch on the compiler and you would just say, "Compile it for parallel" and then it would run faster and automatically be parallel. That's just never going to happen. People have tried and it really doesn't work with the kind of imperative programming styles that we do in mainstream languages like C++ and C# and Java. Those languages are very hard to parallelize automatically because people rely heavily on side effects in their programs.*

Oh, side effects you say? Now *that* explains a lot why some mainstream modern languages started featuring functional programming constructs during the 2000s, like lambdas and closures: C++, PHP, C#, Java, Objective-C, *und so weiter*. One of the key ideas to make mainstream programming languages *parallelizable* was to encapsulate logic in closures and, yes, to avoid side effects.

Abelson and Sussman were nodding all along:

> *Modeling with objects is powerful and intuitive, largely because this matches the perception of interacting with a world of which we are part. However, as we've seen repeatedly throughout this chapter, these models raise thorny problems*

> *of constraining the order of events and of synchronizing multiple processes. The possibility of avoiding these problems has stimulated the development of* functional programming languages, *which do not include any provision for assignment or mutable data. (…) The functional approach is extremely attractive for dealing with concurrent systems.*

("Structure and Interpretation of Computer Programs, Second Edition", by Harold Abelson and Gerald Jay Sussman with Julie Sussman, MIT Press, 1996, page 355.)

In the rush to help programmers use those multicore Intel CPUs, Apple did two things: it added lambdas to Objective-C (but in true Apple style, they were called "blocks"), and created something called Grand Central Dispatch[31] where you could push blocks into queues and have them execute concurrently. But the syntax of Objective-C blocks was unwieldy[32], to put it in a politically correct[33] way, so a few years later they came up with Swift, a new syntax for closures[34], and a new idea for concurrency[35]. Let us rewrite all the wheels every 10 years or so!

Meanwhile, Microsoft (well, actually Anders Hejlsberg) started adding `async` and `await` to all of their languages (mostly C# and TypeScript). But this approach pushed complexity to the programmers[36], and it is not entirely optimal.

Among functional programming languages, there is one in particular that stood out: Erlang[37]. Simply because, well, it was designed to be concurrent from day one.

The publication in 2007 of "Programming Erlang" by the late Joe Armstrong sent shockwaves throughout the industry, back in the days when the Twitter Failwhale[38] was a common daily sight, and everybody blamed Ruby on Rails for it:

> *If we want to write programs that behave as other objects behave in the real world, then these programs will have a concurrent structure.*
> *(…)*
> *Erlang processes have no shared memory. Each process has its own memory. To change the memory of some other process, you must send it a message and hope that it receive and understands the message.*

("Programming Erlang: Software for a Concurrent World", by Joe Armstrong, Pragmatic Programmers, 2007, pages 129 and 130.)

Things that send messages to one another. Alan Kay would agree[39].

Erlang (strongly influenced by CSP) had such a strong impact that by 2014 the team of 32 engineers behind WhatsApp was serving 450 million users[40] with it. The following year they added 18 more engineers[41] to serve 900 million users. Tip: remember to use Erlang the next time you want to sell a startup to Facebook for 19 billion dollars. Also: remember to invite me to the Bahamas after you do that.

Time To Go

All of this is very nice, but Google was still in need of more and more concurrency during their meteoric growth in the 2000s, and neither MapReduce[42] nor Borg[43] were enough.

In October 30th, 2009, a small team at Google composed by Rob Pike[44] (of UTF-8 and Plan 9 fame), Ken Thompson[45] (yes, *that* Ken Thompson, of Unix and C fame), and Swiss computer scientist Robert Griesemer[46] (of V8 JavaScript engine fame) took some cues from CSP and their own (fabulous) past experiences, and presented a new programming language called Go[47]. (The video of the first presentation of Go in public is, by the way, this month's Vidéothèque article[48].)

To make a long story short, Go experienced a meteoric rise that few other languages have seen during their existence. To use a bad reference point, let us just say that it was named twice TIOBE language of the year twice; once in 2009 (because hype) and 2016 (because cloud, more on this later.)

Said growth was nothing short of spectacular, but it was also expected. Go was the first and quintessential "modern" programming language[49], featuring many ideas that were quite revolutionary at the time, but which sound boring[50] and "normal" nowadays. Let us enumerate a few: type inference (thanks to the := operator), fully open source and cross-platform, with closures, with a (very) fast compiler generating (very) efficient code, without semicolons, with mandatory trailing comma for lists and arrays, with multi-line strings, featuring `if` statements without brackets (something that Swift would copy a few years after), bundled with built-in unit

tests (just like the D programming language[51]), and with an integrated package manager (`go get`), a font[52], a mascot[53], and a full library of algorithms ready to use, usually referred to as the "batteries included" approach.

(Catches breath.)

Go, just like C# and Ruby, checked all the marks in the field of developer experience[54]. The Go compiler not only was fast, it also allowed for cross-compilation (allowing Linux developers to create Windows `*.exe` files if required), and even better, it generated quite small binaries. It had a nice website from day one, including a tutorial and a tour of the language[55] that does not require you to install anything on your machine, and got a book authored by Brian Kernighan[56] himself in 2016. The IDE support is also superb these days, ranging from Visual Studio Code[57] to JetBrains GoLand[58] to Vim[59].

But Go also had its share of controversial ideas: to begin with, it used a garbage collector, and that made quite a developer cry in despair. It had pointers! Oh, my gawd! It had structs, but no inheritance! How dare they! And version 1.0 (released in 2012) did not feature generics. What were they thinking?

(Heck, even the name "Go" was controversial[60] at first, as another developer had created a programming language with the same name[61] a few years before. This first conflict even made some headlines[62] around the web. "Don't be evil", they said.)

Maybe even more sacrilegious, the `gofmt` removed a source of conflict in teams (that is why this feature is sacrilegious, by the way; developers love conflicts) and forced the same style for all Go codebases in the past 15 years. The backlash against the language was (and still is, to be honest) epic.

Some kept on whining about generics support, which they finally got it in version 1.18, released in 2022 – arguably the biggest change[63] in the language in over a decade.

Impact

But let us be honest. The cherry on top of the cake was that Go supported a flavor of CSP-inspired, message-based concurrency off-the-box, but with a bonus:

a curly-bracket syntax closer to C than that of Erlang (something that Elixir[64] is trying to correct these days).

Two features enable said concurrency: the `go` keyword, and channels. The language natively allows developers to spawn lightweight processes just by writing its code as just another function, and prefixing their invocation with the `go` keyword.

To exchange data between those lightweight processes, just create a (typed) channel (a simple `c := make(chan int)` would suffice) between them. Just pass the channel as a function argument, use the arrow to pipe data into it, and you are done. Simple and intuitive. Even simpler than sprinkling your code with `async` and `await` keywords, and, needless to say, immensely simpler than using pthreads or other “classic” concurrency primitives.

Large Go programs become swarms of small lightweight processes exchanging little messages with one another; those processes might be in another core in the same computer, or maybe in the same thread; the developer does not need to know exactly how this partition happens (and, to be honest, maybe they do not want to know, at all).

The rest, as they say, is history. Go even found its “killer app” in the world of Cloud Native application development, and has been used to create more than 75% of all projects hosted by the Cloud Native Computing Foundation; to name a few: Kubernetes[65], etcd[66], Helm[67], CockroachDB[68], Grafana Loki[69], Podman[70], Moby[71], Skopeo[72], Kyverno[73], OKD[74], Rancher[75], ZITADEL[76], K9s[77], Caddy[78], Knative[79], Prometheus[80], Colima[81], K3s[82], K3d[83], Minikube[84], CRC[85], Microshift[86], Kind[87]…

The interesting thing about Kubernetes being built with Go is that, from many points of view, Kubernetes could be seen as an operating system to run web services. In 25 years we have gone from a single web server with a single CPU running a monolithic web applications, to Kubernetes clusters of computers (or “nodes”) running various independent processes in parallel (or “pods”), exchanging messages with one another (usually HTML or JSON over HTTP). The scale has changed, the underlying ideas have not.

Beyond the Kubernetes galaxy, Go has found a safe haven in the world of web programming in general. Gin[88], Revel[89], and gorm[90] are popular choices for web apps, while Hugo[91] is a *de facto* standard to build static websites (just like the one you are reading right now, by the way). Google Cloud is built with Go[92] (cannot say I am surprised). More surprising, RoadRunner[93] and FrankenPHP[94] want to provide PHP developers with faster runtimes, in a rare display of solidarity (or pity, you decide).

The web and the cloud are not the only places where Go shines: PocketBase[95], CoreDNS[96], Ollama[97], Oh My Posh[98], Fleet[99], Gitea[100] (and its fork Forgejo[101]), Restic[102], Gitness[103], Terraform[104], Dropbox[105], and Cobra[106] are the proof that developers have embraced Go to quite an extent (and I am pretty sure to be forgetting quite a few projects along the way). Go ended up inspiring other languages, such as the V programming language[107] (which prompts the question, how many letters are still left in the alphabet to name languages?) If all of this is not enough, know that you can even run Go programs with a shebang[108] on your Unix system. How about that.

Triumph

Java[109] was the garbage-collected programming language that begat the first generation of web applications and servers, sometimes having to resort to green threads[110] to try to run things in parallel. It was both a language and a runtime that thrived in a world of single-core CPUs and multithreaded web servers.

In contrast, Go appears as a modern heir, featuring many answers to what could be seen as shortcomings in Java; but it is still, in essence, a garbage-collected language that builds standalone, small, native, and fast[111] binaries, with a huge supporting framework baked in and ready to use.

Go is much more than that, though. Go is a triumph in developer experience and efficiency, a language and a runtime created by (needless to say) *very* experienced designers built to solve a particular problem (scalable web services) in a lightweight manner. Just like PostgreSQL[112] and Git[113], some technologies survive Darwinian evolutionary cataclysms and rise to the top of their craft. Without any doubt, Go

belongs to this select group, and if somewhat naïvely we take the past 15 years as a proof, its future looks definitely bright.

And if Superman III was ever to be remade (Note to Hollywood: please do not), Gus Gorman should probably use Go to calculate those "two bilaterial coordinates at the same time".

Cover photo by Lance Grandahl[114] on Unsplash[115].

References

[1] https://en.wikipedia.org/wiki/I386
[2] https://en.wikipedia.org/wiki/Power_Mac_G5
[3] https://en.wikipedia.org/wiki/Herb_Sutter
[4] https://web.archive.org/web/20100210171718/http://www.gotw.ca/publications/concurrency-ddj.htm
[5] https://web.archive.org/web/20130331094524/http://www.drdobbs.com/cpp/the-trouble-with-locks/184401930
[6] https://web.archive.org/web/20130331055234/https://www.drdobbs.com/cpp/the-trouble-with-locks/184401930?pgno=2
[7] https://deprogrammaticaipsum.com/dr-dobbs-journal-of-computer-calisthenics-and-orthodontia/
[8] https://en.wikipedia.org/wiki/Moore's_law
[9] https://en.wikipedia.org/wiki/Pentium_FDIV_bug
[10] https://en.wikipedia.org/wiki/No_such_thing_as_a_free_lunch
[11] https://en.wikipedia.org/wiki/Intel_Core
[12] https://en.wikipedia.org/wiki/Superman_III
[13] https://web.archive.org/web/20201109025630/https://www.denofgeek.com/movies/what-superman-3-teaches-us-about-computer-programming/
[14] https://deprogrammaticaipsum.com/programming-the-liberal-arts/
[15] https://en.wikipedia.org/wiki/Scott_Meyers
[16] https://en.wikipedia.org/wiki/C%2B%2B11
[17] https://deprogrammaticaipsum.com/the-art-of-the-art-of-computer-programming/
[18] http://sortbenchmark.org/
[19] https://en.wikipedia.org/wiki/Barry_Boehm
[20] https://viterbischool.usc.edu/news/2022/09/barry-boehm-a-living-legend-in-systems-and-software-engineering-dies-at-87/
[21] https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf
[22] https://en.wikipedia.org/wiki/Cilk
[23] https://en.wikipedia.org/wiki/OpenMP
[24] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[25] https://en.wikipedia.org/wiki/Tony_Hoare
[26] https://dl.acm.org/doi/10.1145/359576.359585
[27] https://www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2/
[28] https://deprogrammaticaipsum.com/feeling-lucky/
[29] https://deprogrammaticaipsum.com/where-does-microsoft-want-to-go-today/
[30] https://en.wikipedia.org/wiki/Anders_Hejlsberg
[31] https://en.wikipedia.org/wiki/Grand_Central_Dispatch
[32] http://fuckingblocksyntax.com/
[33] http://goshdarnblocksyntax.com/
[34] http://www.goshdarnclosuresyntax.com/

[35] https://www.swift.org/documentation/concurrency/
[36] https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/
[37] https://en.wikipedia.org/wiki/Erlang_(programming_language)
[38] https://www.theatlantic.com/technology/archive/2015/01/the-story-behind-twitters-fail-whale/384313/
[39] https://deprogrammaticaipsum.com/the-absolute-no-frills-quite-ignorant-very-incomplete-and-certainly-flawed-beginners-guide-to-smalltalk/
[40] https://www.fastcompany.com/3026758/inside-erlang-the-rare-programming-language-behind-whatsapps-success
[41] https://www.wired.com/2015/09/whatsapp-serves-900-million-users-50-engineers/
[42] https://en.wikipedia.org/wiki/MapReduce
[43] https://en.wikipedia.org/wiki/Borg_(cluster_manager)
[44] https://en.wikipedia.org/wiki/Rob_Pike
[45] https://en.wikipedia.org/wiki/Ken_Thompson
[46] https://en.wikipedia.org/wiki/Robert_Griesemer
[47] https://en.wikipedia.org/wiki/Go_(programming_language)
[48] https://deprogrammaticaipsum.com/rob-pike/
[49] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[50] https://www.capitalone.com/tech/software-engineering/go-is-boring/
[51] https://akos.ma/blog/d-or-what-go-may-have-been/
[52] https://go.dev/blog/go-fonts
[53] https://go.dev/blog/gopher
[54] https://eltonminetto.dev/en/post/2024-06-12-go-is-a-plataform/
[55] https://go.dev/tour/welcome/1
[56] https://deprogrammaticaipsum.com/brian-kernighan/
[57] https://marketplace.visualstudio.com/items?itemName=golang.Go
[58] https://www.jetbrains.com/go/
[59] https://github.com/fatih/vim-go
[60] https://web.archive.org/web/20091223042821/http://code.google.com/p/go/issues/detail?id=9
[61] https://web.archive.org/web/20091114092929/http://en.wikipedia.org/wiki/Go!_(programming_language)
[62] https://web.archive.org/web/20091113084340/http://www.informationweek.com/news/software/web_services/showArticle.jhtml?articleID=221601351
[63] https://thenewstack.io/go-1-18-the-programming-languages-biggest-release-yet/
[64] https://en.wikipedia.org/wiki/Elixir_(programming_language)
[65] https://kubernetes.io/
[66] https://etcd.io/
[67] https://helm.sh/
[68] https://www.cockroachlabs.com/
[69] https://grafana.com/oss/loki/
[70] https://podman.io/

[71] https://mobyproject.org/
[72] https://github.com/containers/skopeo
[73] https://kyverno.io/
[74] https://www.okd.io/
[75] https://www.rancher.com/
[76] https://zitadel.com/
[77] https://k9scli.io/
[78] https://github.com/caddyserver/caddy
[79] https://knative.dev/
[80] https://prometheus.io/
[81] https://github.com/abiosoft/colima
[82] https://github.com/k3s-io/k3s/
[83] https://github.com/k3d-io/k3d
[84] https://github.com/kubernetes/minikube
[85] https://crc.dev/blog/
[86] https://github.com/openshift/microshift
[87] https://kind.sigs.k8s.io/
[88] https://gin-gonic.com/
[89] https://revel.github.io/
[90] https://gorm.io/
[91] https://gohugo.io/
[92] https://cloud.google.com/go
[93] https://roadrunner.dev/
[94] https://frankenphp.dev/
[95] https://pocketbase.io/
[96] https://coredns.io/
[97] https://ollama.com/
[98] https://ohmyposh.dev/
[99] https://fleetdm.com/
[100] https://about.gitea.com/
[101] https://forgejo.org/
[102] https://restic.net/
[103] https://gitness.com/
[104] https://www.terraform.io/
[105] https://dropbox.tech/infrastructure/open-sourcing-our-go-libraries
[106] https://cobra.dev/
[107] https://vlang.io/
[108] https://stackoverflow.com/a/42865595
[109] https://deprogrammaticaipsum.com/write-anywhere-run-once/
[110] https://en.wikipedia.org/wiki/Green_thread
[111] https://benhoyt.com/writings/go-version-performance-2024/

[112] https://deprogrammaticaipsum.com/the-elephant-in-the-room/
[113] https://deprogrammaticaipsum.com/twenty-years-is-nothing/
[114] https://unsplash.com/@lg17?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash
[115] https://unsplash.com/photos/brown-metal-train-rail-near-rocky-mountain-during-daytime-nShLC-WruxQ?utm_content=creditCopyText&utm_medium=referral&utm_source=unsplash

Rob Pike

By Adrian Kosmaczewski

This is the story of four videos by Rob Pike[1], one of the creators of the Go programming language, each marking important milestones throughout the past 15 years: from the introduction of the language to the world in 2009, to its time of meteoric growth during 2012 and 2015, and finally to a review of the first 14 years of existence in 2023.

2009

The first video[2] is the introduction of the Go programming language at Google on October 30th, 2009. This video is part of the Google TechTalks series we have talked about in a previous article[3] of this magazine.

There are various interesting aspects to this video, starting with the opportunity that web video brings as a store of recent historical references; where are the introduction videos of languages such as FORTRAN, C++, or even PL/I? Going back in time provides, such in this case, the possibility of understanding the *why* of the language, the reasons and the needs that it fulfilled, which in turn gives quite a bit of background to *l'air du temps* at the time of its birth.

Rob Pike starts the presentation calling Go "experimental", setting expectations up front, in a clear example of what I call "honestization"[4], but in a vain effort to contain the inevitable hype[5] that surrounded the language. (And hype around Go there was, even a lot.)

The gist of the situation in 2009 was clear and dramatic: the new world was all about computers that have stopped getting faster (or, as Herb Sutter said, the free lunch was over), lots of networking, the rise of computer clusters, the need for scalability, and the dramatic destruction of developer productivity generated by slow build cycles.

The tension between strong typing and long compiler cycles was dragging the whole industry down. According to Pike,

> *You can be productive or safe, not both.*

The backlash against Java around that time (which we discussed in a previous edition[6] of this magazine) was strong, and translated in the rebirth of dynamic programming languages (with Ruby, JavaScript, and Python leading the charge). But as Robert Griesemer said (quoted by Rob Pike),

> *Clumsy type systems drive people to dynamically typed languages.*

The idea of Go was from the beginning to find an equilibrium and to provide a decent type system that would also yield a good developer experience. In contrast, Rust chose to ditch that requirement, pushing the envelope of what a type system could do to the extreme, at the expense of compile times.

Go became, then, an object-oriented language without type hierarchies, with a clean and concise syntax, with zero-initialized variables, a garbage collector based on the IBM Recycler[7] design, featuring concurrency with channels, interfaces, and a strong compatibility promise[8].

But what did Rob Pike say about generics support? Well,

> *Would definitely be useful: in short, not yet.*

Sorry people, you'll have to wait until the 2020s for that.

2012

Comes next "Concurrency is not Parallelism"[9], a talk by Rob Pike given at the Waza 2012 conference, precisely the same year when Go version 1.0 was released[10].

This talk starts with an appropriate observation:

> *You might think the world is object-oriented, but it's not, it's concurrent.*

But as the title suggests, Pike explains that concurrency is not exactly the same as parallelism; actually, concurrency is much more than just programming code that runs on multiple CPUs at once.

Concurrency, for Rob Pike, is a way to think about programs, as small independent processes that do their own thing, communicating when needed, reporting back when they are done. Concurrency is about structure and composition.

The architectural ideas embedded in Go (as well as those in Erlang and Crystal[11]) were originally explained by Tony Hoare in his "Communicating Sequential Processes"[12] 1978 paper.

Parallelism, on the other hand, is about the execution itself, when various parts of a program happen to be running at the same chronological time. There is no structure implied in this concept. We don't have to worry about parallelism when doing concurrency: a design might be concurrent, but not work in a parallel way. The issue is to design solutions as concurrent, not as parallel, per se.

Developers must not think about running things in parallel, but instead should concentrate in breaking the problem into things that can run concurrently, doing different tasks at once. Concurrency is not parallelism, but enables it.

For those unfamiliar with the language (this was 2012 after all, the year Go went 1.0) Rob Pike provides a short intro to the language around 14:00, in particular, comparing Go coroutines with the ampersand on the Linux shell. To provide some context, Rob Pike also links to a paper by Russ Cox[13] that connects the historical dots between Go (among other languages) and communicating sequential processes (CSP) as described by Tony Hoare.

Rob Pike's phrase at 05:40 appropriately summarizes the talk:

> *These ideas are not deep, they are just good.*

Maybe what is not *that* good is to use "Gophers burning C++ books concurrently" as an example program to demonstrate this way of thinking. Can we just stop using these backward and passive-aggressive kind of language?

2015

Rob Pike's "Go Proverbs"[14] presentation at Gopherfest 2015 is our third video. From a historical point of view, this was the time of the rise of containers and Kubernetes, and with them, the rise in prominence of Cloud Native technologies and microservices as a dominant architectural paradigm.

It turns out that the name of the programming language is not entirely unrelated to the game of the same name[15], in particular because both feature easy rules that are hard to master. Pushing the analogy a bit too far, Rob Pike takes inspiration from the 1960 book "Go Proverbs Illustrated" by Kensaku Segoe and its list of traditional go proverbs[16] to state his own list of 18 proverbs for the programming language.

(This is an exercise not too far removed from that of Geoffrey James' "Tao of Programming" which, by the way, we have discussed in a previous issue[17] of this magazine.)

Without boring the reader, here are some of the most salient of Pike's Go Proverbs:

> *Don't communicate by sharing memory, share memory by communicating.*
> *(…)*
> *Make the zero value useful.*
> *(…)*
> *A little copying is better than a little dependency.*
> *(…)*
> *Clear is better than clever.*
> *(…)*
> *Don't just check errors, handle them gracefully.*

Needless to say, many of the 18 proverbs apply not only to Go, but to pretty much every programming language you can think of. Precisely about this point, there is an interesting observation of Pike toward the end of the video:

> *Go created a community around a certain way of programming*

2023

Finally, the fourth video[18] of this edition is Pike's "What We Got Right, What We Got Wrong" talk at GopherConAU 2023, coinciding with Go's 14th birthday.

Let us start with the things that Pike is happy about: the formal specification and the portability of the language; its compatibility from 1.0 onward; the tooling and libraries provided since day one, in particular `gofmt`; interfaces, which are Pike's favorite feature; and the support for concurrency (of course!)

> *Goroutines in Go, coroutines in Lua, and fibers in Ruby are perfectly adequate.*

What was not so good? Engaging with the community took some time (and friction) to get right; the process of developing the package manager was suboptimal; and the quality and quantity of documentation and examples, at least at the beginning, left a lot to be desired.

What about generics? According to Pike, “Generics” is the wrong word for what Go ended up providing: “parametric polymorphism” is a better description.

In general, the original goal of the project was not to create a new programming language, but rather to create a new way of programming large production systems with large teams. In this sense, yes, the project can be considered a resounding success.

This session ends with a somewhat terrible question and answer session at the end, to which Pike tries to answer in the best possible manner, providing some interesting cues at the same time:

> *(35:21) There will be no such thing as Go 2 in the foreseeable future.*
> *(36:07) One wish, a feature we should have done but didn’t: arbitrary precision integer types.*
> *(41:10) Anything that makes your program safer at compile time it’s good, but I don’t want to spend 2 hours waiting for it to build.*

More Videos

There are plenty of excellent videos around the Go programming language, but if you are hungry for more, here go some suggestions. First, two videos by Russ Cox: “Go Changes”[19] at GopherCon 2023, and “Compatibility: How Go Programs Keep Working”[20] at GopherCon 2022.

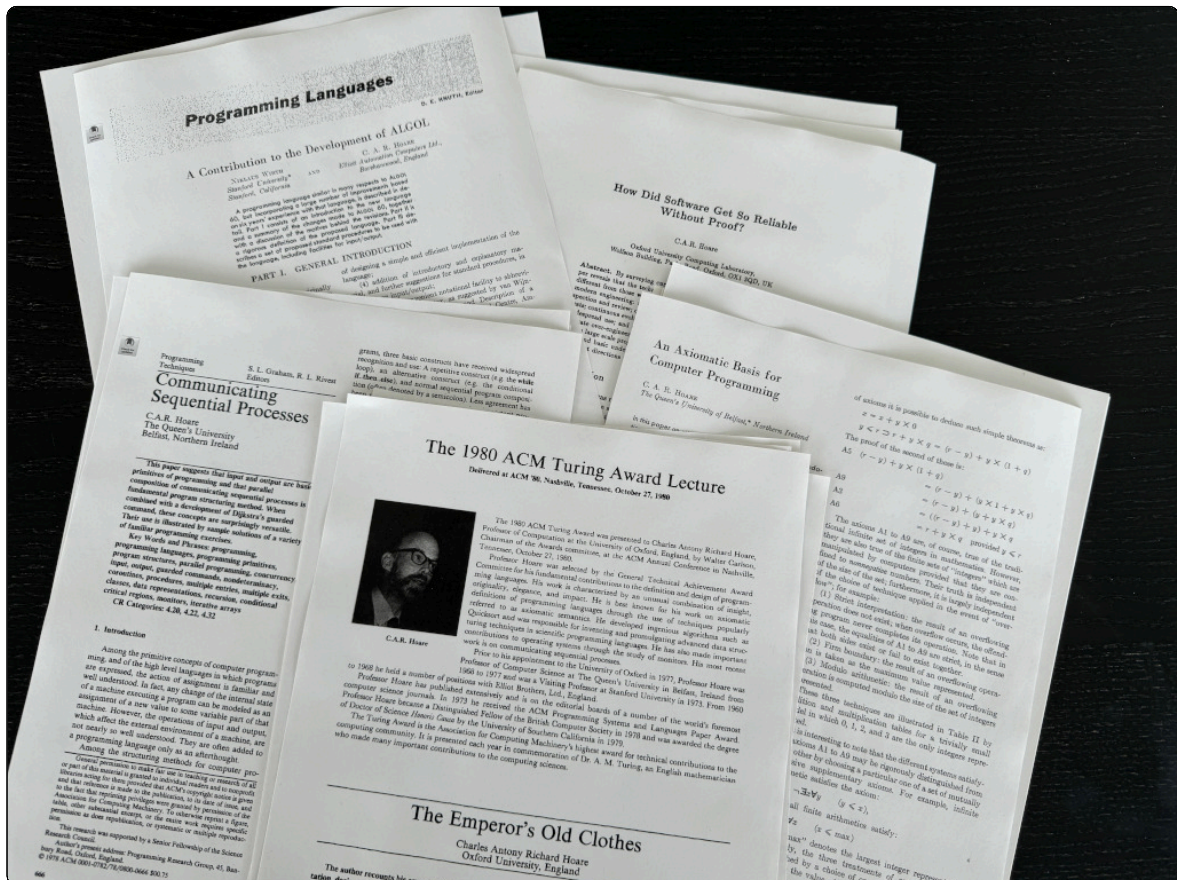
Finally, the excellent “Go in 100 Seconds”[21] on the Fireship channel, which we covered a few months ago in another article[22] in this magazine.

Cover snapshot chosen by the author from the first video.

References

[1] https://en.wikipedia.org/wiki/Rob_Pike
[2] https://www.youtube.com/watch?v=rKnDgT73v8s
[3] https://deprogrammaticaipsum.com/google-techtalks/
[4] https://deprogrammaticaipsum.com/less-evangelization-more-honestization/
[5] https://deprogrammaticaipsum.com/mainstream-is-the-new-hype/
[6] https://deprogrammaticaipsum.com/write-anywhere-run-once/
[7] https://web.archive.org/web/20091114151556/http://www.research.ibm.com/people/d/dfb/papers.html
[8] https://go.dev/doc/go1compat
[9] https://www.youtube.com/watch?v=oV9rvDllKEg
[10] https://go.dev/doc/go1
[11] https://crystal-lang.org/
[12] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[13] https://swtch.com/~rsc/thread/
[14] https://www.youtube.com/watch?v=PAAkCSZUG1c
[15] https://en.wikipedia.org/wiki/Go_(game)
[16] https://en.wikipedia.org/wiki/Go_proverb
[17] https://deprogrammaticaipsum.com/geoffrey-james/
[18] https://www.youtube.com/watch?v=yE5Tpp2BSGw
[19] https://www.youtube.com/watch?v=BNmxtp26I5s
[20] https://www.youtube.com/watch?v=v24wrd3RwGo
[21] https://www.youtube.com/watch?v=446E-r0rXHI
[22] https://deprogrammaticaipsum.com/fireship/

Sir Tony Hoare



By Adrian Kosmaczewski

It would be unwise and useless to try to summarize in a thousand words the immense contributions of Sir Charles Antony Richard Hoare[1], also known as Tony Hoare (I suppose we are all good friends in this industry) or, with a more Tolkien feeling, as C. A. R. Hoare. I will settle for "Sir Tony Hoare" in this article; familiar yet respectful enough.

I will try to focus here on the concurrency part, and its obvious contribution to the Go programming language. Suffice to say that when you have Quicksort[2], communicating sequential processes[3], a billion-dollar mistake[4], the Hoare Triple[5],

the Hoare program correctness logic, the dining philosophers problem, a knighthood, and a Turing Award in your résumé, you hardly need an introduction, least of all by me.

(To add more gravitas to our feeling of underachievement, let us remember that Sir Tony Hoare spoke Russian, studied with Andrey Kolmogorov[6] in the Soviet Union, helped implement ALGOL, taught computer science at Oxford, worked as researcher for Microsoft, and was named a Fellow of the Royal Society (actually, maybe I should add the "FRS" suffix to the name, too.) Did I mention that he is also thoroughly considered a kind human being?)

Two years before receiving the Turing Award, Sir Tony Hoare published a seminal paper called "Communicating Sequential Processes"[7] commonly referred to as "CSP". The fact that there is a Wikipedia page[8] dedicated to the concept should give you some context about the "ground-breakingness" of this paper.

In short, and as Hoare himself explains in an interview[9], the experience of a failed multiprocessing project took him to elaborate on the complete and absolute banishment of shared state as a *conditio sine qua non* for process communication; instead, he argued, processes should be independent, isolated, but able to communicate with one another in a formal way.

Yes, pretty much what the Go syntax proposes today. It even borrowed the arrow (prominently used all along Sir Hoare's paper, representing the various exchanges among parties) as a syntactic element, even though the version we got in the language points to the left, instead of pointing to the right (which in my mind still looks odd, but maybe that is because I am a lefty.)

The question that appears in my mind as I read the paper is, why did Go only appear in 2009? Well, Erlang[10] implemented some of these ideas as back as 1986, so clearly somebody at Ericsson was busy reading important computer science papers. Actually, another programming language you might have not heard about, occam[11], implemented these ideas even earlier, in 1983.

The ideas behind CSP were expanded in the 1985 book of the same name, a book that is freely available to download[12] today.

If you are interested in learning more about Sir Tony Hoare, check the 2021 book "Theories of Programming: The Life and Works of Tony Hoare"[13] by Cliff B. Jones and Jayadev Misra, and published by the Association for Computing Machinery. Sir Hoare's book "Structured Programming" co-authored with Ole Dahl[14] and Edsger Dijkstra[15] is freely available on the Internet Archive[16].

You can also read the July 2024 edition[17] of the Newsletter of the Formal Aspects of Computing Science (FACS) Specialist Group[18], which was dedicated to Sir Tony Hoare in his 90th birthday, featuring articles from various experts who worked with him.

Alternatively, you can also peruse the various papers shown in the cover image of this article:

- "A contribution to the development of ALGOL"[19], Communications of the ACM, volume 9, issue 6, pages 413-432, 1966, authored with none other than Niklaus Wirth[20] himself.
- "An axiomatic basis for computer programming"[21], Communications of the ACM, volume 12, issue 10, pages 576-580, 1969.
- "Communicating sequential processes"[22], Communications of the ACM, volume 21, issue 8, pages 666-677, 1978.
- "The emperor's old clothes"[23], Communications of the ACM, volume 24, issue 2, pages 75-83, 1980. This was Hoare's Turing Award Lecture, delivered at ACM '80 in Nashville, Tennessee, USA.
- How Did Software Get So Reliable Without Proof?[24], FME '96: Industrial Benefit and Advances in Formal Methods, Third International Symposium of Formal Methods Europe, Co–Sponsored by IFIP WG 14.3, Oxford, UK, March 18–22, 1996. Proceedings edited by Marie–Claude Gaudel and Jim Woodcock, pages 1-17.

Reading About Go

Satisfying the curiosity of those who came to this page looking for books about Go, here go some recommendations.

- We have already talked about Kernighan's book[25] in this magazine.

- The Go language website contains a short tour[26] that might be useful to newcomers. There is also a useful "Effective Go"[27] document you might want to check.
- For those looking to learn in a step-by-step basis, "Practical Go Lessons"[28] and "Learn Go with Tests"[29] will prove to be foundational.
- If you already know Go and instead would like to see how to use it in a real project, I can only recommend "Writing an interpreter in Go"[30] and "Writing A Compiler In Go"[31] by Thorsten Ball.
- There is a free chapter about Generics[32] available online, extracted from a classic book that recently got its 2nd edition[33].
- And if all of this was not enough, there are more[34] and more[35] books for your reading pleasure.

Cover photo by the author.

References

[1] https://en.wikipedia.org/wiki/Tony_Hoare
[2] https://en.wikipedia.org/wiki/Quicksort
[3] https://www.youtube.com/watch?v=QUOlyIHmBrM
[4] https://www.youtube.com/watch?v=ybrQvs4x0Ps
[5] https://deprogrammaticaipsum.com/how-to-reason-about-mutable-state/
[6] https://en.wikipedia.org/wiki/Andrey_Kolmogorov
[7] https://www.cs.cmu.edu/~crary/819-f09/Hoare78.pdf
[8] https://en.wikipedia.org/wiki/Communicating_sequential_processes
[9] https://www.youtube.com/watch?v=QUOlyIHmBrM
[10] https://en.wikipedia.org/wiki/Erlang_(programming_language)
[11] https://en.wikipedia.org/wiki/Occam_(programming_language)
[12] http://www.usingcsp.com/
[13] https://dl.acm.org/doi/book/10.1145/3477355
[14] https://en.wikipedia.org/wiki/Ole-Johan_Dahl
[15] https://en.wikipedia.org/wiki/Edsger_W._Dijkstra
[16] https://archive.org/details/Structured_Programming__Dahl_Dijkstra_Hoare
[17] https://www.bcs.org/media/1wrosrpv/facs-jul24.pdf
[18] https://www.bcs.org/membership-and-registrations/member-communities/facs-formal-aspects-of-computing-science-group/newsletters/back-issues-of-facs-facts
[19] https://dl.acm.org/doi/10.1145/365696.365702
[20] https://deprogrammaticaipsum.com/niklaus-wirth/
[21] https://dl.acm.org/doi/10.1145/363235.363259
[22] https://dl.acm.org/doi/10.1145/359576.359585
[23] https://dl.acm.org/doi/10.1145/358549.358561
[24] https://www.cs.ox.ac.uk/publications/publication8320-abstract.html
[25] https://deprogrammaticaipsum.com/brian-kernighan/
[26] https://go.dev/tour/welcome/1
[27] https://go.dev/doc/effective_go
[28] https://www.practical-go-lessons.com/
[29] https://quii.gitbook.io/learn-go-with-tests
[30] https://interpreterbook.com/
[31] https://compilerbook.com/
[32] https://learning-go-book.dev/chapter15_learningGo.pdf
[33] https://www.oreilly.com/library/view/learning-go-2nd/9781098139285/
[34] https://github.com/dariubs/GoBooks
[35] https://go.dev/wiki/Books