REFPERSYS high-level goals and design ideas*

Basile STARYNKEVITCH† Abhishek CHAKRAVARTI‡
Nimesh NEEMA§
refpersys.org

October 2019 - February 2020

Abstract

REFPERSYS is a **REF**lexive and orthogonally **PER**sistent **SYS**tem (as a GPLv3+ licensed free software[1]) running on Linux; it is a hobby[2] but serious **research project** for many years, mostly aimed to experiment **open science** ideas close to Artificial General Intelligence[3] dreams, and we don't expect useful or interesting results before several years of hard work.

audience : LINUX free software developers[4] and computer scientists interested in an experimental open science approach to reflexive systems, orthogonal persistence, symbolic artificial intelligence, knowledge engines, etc....

Nota Bene: this report contains many hyperlinks to relevant sources so its PDF should rather be read on a computer screen, e.g. with `evince`. Since it describes a circular design (with many cycles [21]), we recommend to read it twice (skipping footnotes and references on the first read).

---

*This document has git commit `4d33386d51490039`, was Lua-LaTeX generated on *2020-Feb-13 15:10 MET*, see `gitlab.com/bstarynk/refpersys/` and its `doc/design-ideas` subdirectory. Its draft is downloadable, as a PDF file, from `starynkevitch.net/Basile/refpersys-design.pdf` ...

†See `starynkevitch.net/Basile/` and contact `basile@starynkevitch.net`, 92340 Bourg La Reine (near Paris), France.

‡`chakravarti.avishek@gmail.com`, FL 3C, 62B PGH Shah Road, Kolkata 700032, India.

§`nimeshneema@gmail.com`, 206 Sundaram Apartment, 38/2, Biyabani, Indore 452002, India.

[1]Some code is available on `gitlab.com/bstarynk/refpersys`.

[2]Basile Starynkevitch (France) wants to find some research grant funding related to this. Please mention potential funding opportunities (call for research project proposals) by email to `basile@starynkevitch.net`.

[3]Artificial General Intelligence

[4]Those LINUX software developers are routinely *glancing inside*, *building* then using -from their published source code- quite large open source programs (such as GCC, SBCL, CHICKEN-SCHEME, HOP, HAXE, OCSIGEN, EMACS, SQLITE, MARIADB, etc...) and perhaps even contributing to smaller free software projects like NINJA, `libonion`, etc... By the way, all these open source projects could be useful to or inspirational for REFPERSYS.

Contents

1 Social Necessity of AGI Systems with Long Term Development

Our complex, but fragile, world is facing dramatic and extremely challenging planet-wide issues, such global warming, demographic and political crises, economic and financial emergencies, and growing inqualities. In the light of such challenges, **A**rtificial **G**eneral **I**ntelligence (*AGI*) systems are increasingly relevant. @@TODO: explain how?

As the slow, progressive Darwinian evolution of human intelligence shows, the limited intelligence of the *Homo Sapiens* [5] species took more than a million years (about 30,000 generations) to continually evolve from an ape-like state.

Our observation of natural human intelligence (which has not yet been fully understood or modelled[6]) has led us to believe that there is no single, simple model of intelligence. Similarly, any AGI system must necessarily have a very complex and self-improving organisation.

We are aware than any progress towards AGI will be slow (many years, perhaps decades[7]) and progressive. Remember Hofstadter's Law: *"It always takes longer than you expect, even when you take into account Hofstadter's Law"* [21] and Brook's observations [8, 9] that *"if one woman can give birth in 9 months, 9 women cannot give birth to a baby in one month"*. For "giving birth" to REFPERSYS, a small team could need at least 9 years. However, intermediate results or side effects are not predictable but could be useful even during the REFPERSYS project.

We believe in free software (read also this), and we strongly believe that an AGI prototype should be some free software, exactly like most infrastructure software are (notably LINUX). See also the SOFTWARE HERITAGE project for interesting insights. REFPERSYS wants to be an AGI infrastructure , and there is work for many years (several years of work needed without any "artificial intelligence", just for the infrastructure).

An even partially successful AGI system might be useful to coordinate, run and manage other existing software (described through some knowledge given declaratively). Imagine how complex future digital twins of the entire planet Earth, designed

[5] In Latin, *Homo Sapiens* means "the human who knows what it knows" and, interestingly enough, relates to both metaknowledge and Reflection.

[6] Half a billion euros of European taxpayers' money were spent on the Human Brain Project, but did not lead to a complete, reproducible, artificial model of human intelligence; of course, it did fund interesting and successful research!

[7] An interesting parallel could be controlled nuclear fusion -which also bears some "bootstrrapping" concepts- with ITER; we expect REFPERSYS to cost several thousand times less at least; but even partial AGI success is as important for humanity as nuclear fusion produced electricity, and a future REFPERSYS might even help that ITER megaproject or other ones.

to tackle with global warming, would need to be. For such dramatically complex usage, an AGI system (like REFPERSYS, if we succeed in making it) could be quite helpful to just drive and use such a "digital twin" simulation. Making it free software runnable on a free software operating system should benefit most of humanity (but keeping it proprietary won't), and enable further or alternative experimentations. And "there is no planet B"[8]. So investing a few persons willing to working for nearly a decade is not too much for such a perspective.

2 REFPERSYS ambitions and goals

2.1 REFPERSYS core idea[l]?s

The title of this subsection is *not* a typo[9]. We indeed mean both *ideas* (that is, software design and architectural concepts, guiding our daily implementation efforts) and *ideals* (that is, long term research objectives and ambitions).

The REFPERSYS[10] system shares several -but not all- goals and design ideas (but no code) with `bismon` [50] but of course *not* `bismon`'s application[11] to static source code analysis. Like `bismon`, REFPERSYS is a **reflexive** (it uses reflection), **introspective** and **orthogonally persistent** system, but not for static program analysis. Please read Bismon's draft report [50] for a more precise definition of these concepts. **REFPERSYS is a long term[12] risky research project with an open science mindset and reproducible experiment ethics [54, 34], and a free software licensed under GPLv3+, and targetted *only* for LINUX X86-64 computers.**. A Linux system[13] with at least 16 Gibytes of RAM, 4 *x86-64* cores, and 220 Gibytes of disk is required. The grand ambition of REFPERSYS is to become later an infras-

[8] As reminded E.Macron, president of France, to the US Congress.

[9] It is a geeky pun on words with shell globbing and regexpr like syntax.

[10] For a **Ref**lexive **Per**sistent **Sys**tem

[11] I Basile am not allowed and not funded to directly work on AGI -which still is my major personal scientific interest- but I do get funded on applied research projects like DECODER and try to push some AGI ideas into them.

[12] I don't expect any significant AGI research results before ≈ 2026.

[13] My own `ours.starynkevitch.net` computer, running *Debian/Unstable*, has 64 Gibytes of RAM, 24 cores (AMD 2970WX) and terabytes of disk space, including a terabyte of SSD.

tructure for some strong AGI system à la CAIA[14] by Jacques Pitrat[15] [37, 35, 36], but before even approaching that goal a big lot of work is required, and REFPERSYS should be valuable by itself for other less ambitious and more pragmatical purposes, perhaps some specialized collaborative web server (GPLv3+) to ease communication between human REFPERSYS developers, that is a mix of a wiki, a chat, and a tool for sharing document with drawings or graphics.

The development of REFPERSYS is (like the one of `bismon`, or of CAIA) a slow, incremental and gradual bootstrapping process with a meta-programming [15, 19] approach : features added to REFPERSYS in January 2020 are used to implement new features worked on a later REFPERSYS in March 2020.

As every practical software, REFPERSYS targets some defined machines: common Linux distribution running on some computer[16]. So the target machine of REFPERSYS is a quite complete and modern Linux system (such as a recent DEBIAN or UBUNTU desktop), with many useful packages, and administered by some human person[17]. The REFPERSYS system is published in "source" form, as a set of `git` versioned[18] textual files (e.g. hopefully generated *C* files[19], perhaps some `Makefile` or better yet an OMAKE build -most and more and more[20] of them being generated- or shell files or data files). Some of these files are generated, and the bootstrapping goal is to have *every* `git`-registered textual file been generated by REFPERSYS, with a **bootstrap**ed approach[21] similar to those of self-hosting compilers.

[14]With explicit permission from J.Pitrat, CAIA source code -entirely generated by itself, about half a million lines of C code- is available on my (Basile's) web page as `caia-su-24feb2016.tar.bz2`, and you could build it with `gcc -O -g [A-Z]*.c -rdynamic -ldl` then run `./a.out`. However, since I Basile sadly failed to convince J.Pitrat that open source [29, 52] software are -in our XXIth century- also an important way to transmit research ideas, there are no complete instructions to use it. Hence CAIA has an undocumented user interface as user-friendly as the one of `ed` but convenient enough to J.Pitrat alone! If you are capable of reading some comments in French and guessing the semantics of declarative "expert system" like rules (CAIA has more than a dozen of thousands of them), run it, then type `L EDITE` and start reverse-engineering that brillant CAIA system.

[15]Jacques Pitrat has passed away on October 14th, 2019. See quickly also his old web page on `jacques.pitrat.pagesperso-orange.fr` and his interesting blog on `bootstrappingartificialintelligence.fr/WordPress3`...

[16]For several years, that computer is a desktop or powerful laptop running some DEBIAN. Later that could be some "virtual machine" e.g. some DOCKER container.

[17]For obvious cybersecurity reasons, automatic administration of that Linux distribution is out of scope. Also, since Basile Starynkevitch is still working (in October 2019) in a cybersecurity lab (of about 25 permanent staff) at CEA/LIST, cybersecurity concerns would be a conflict of interest.

[18]We crucially depend upon `git` *specifically* (e.g. `GitLab`), and porting REFPERSYS to some other versioning system -or to some other operating system than LINUX- would be a quite difficult task.

[19]However, notice that bootstrapped language implementations like Scheme 48 or OCaml are keeping some bytecode form under version control, and CHICKEN SCHEME is, like `bismon`, `git`-keeping generated C files.

[20]Of course, in a chicken and egg fashion, the initial version of REFPERSYS has to contain mostly hand-written files!

[21]Observe that Linux source distributions like `linuxfromscratch.org`, or to a lesser extent

Within REFPERSYS, we call[22] "source file" any Linux file which is `git`-versioned. We hope that more and more of these source files will be generated by the `refpersys` ELF executable program. **A significant milestone is the entire bootstrapping of REFPERSYS**, when all files (in textual form, to stay `git`-friendly, like text based protocols are more friendly for developers) can be regenerated by the `refpersys` executable, exactly in the same state as they were previously[23] : as a whole, our REFPERSYS system should become a Quine program, and CAIA is already one. So the build automation tool which compiles REFPERSYS should use file contents, not modification times to trigger compilation commands, since a full regeneration of such a bootstrapped REFPERSYS system will touch all files, without changing the content of any of them. Hence and very concretely, for building REFPERSYS the `OMake` build automation tool is preferable to GNU `make`.

For pragmatical reasons, **REFPERSYS needs a good garbage collector** (or GC [3, 53, 4, 22]), since fully compile-time GC [32] are too difficult to implement. Since multi-core x86-64 machines are very common, it should take advantage of them, so **REFPERSYS should follow a multi-threaded approach** above POSIX [5] or C++11 threads. Our GC should be a precise garbage collector [42] and we may want to favor, like what was done in GCC MELT [49, 48, 47], fast allocation of small memory zones which get quickly disposed of when becoming dead using a copying generational Cheney-like GC algorithm [53]. But mixing precise, sometimes generational GC techniques with multi-threading is a difficult programming task. But precise-GC friendly programming is simpler in generated C or C++ code that with hand-written code (because of explicit management of local GC roots and write barriers, à la QISH or OCAML: garbage collection invariants are boring and brittle to maintain in hand-written code).

Reification is an important concept in REFPERSYS, including (later) at the knowledge representation level with semantic networks and frames. REFPERSYS call stacks are made of call frames known to our garbage collector (like OCAML's

GenToo, are also, when considered as a single system, fully bootstrapped.

[22] Notice that, on purpose, our terminology is different of usual habits in the open source realm: almost all software projects (see also `softwareheritage.org`) are made of *computer files* typed by human developers in some source-code editor or some IDE such as `Emacs`, `vim` or `Code::Blocks`, according to the old Unix philosophy. Notice that large open source projects like the LIBREOFFICE suite, the GCC compiler collection or the FireFox browser tend to accept plugins instead of favoring old fashioned command pipelines, but multi-threaded applications may follow the pipeline design pattern. In contrast, we are impatient to reach the state where all REFPERSYS source files have been `git`-versioned but are all generated by a previous run of our `refpersys` executable. The REFPERSYS developer is interacting, through a web interface, with some running `refpersys` process, which is also some specialized web server (using HTTP).

[23] Pedantically, some fixpoint of some very coarse-grained operational semantics related to abstract interpretation and big step semantics, each big step being the entire regeneration of the system, inspired by Futurama projections and partial evaluation.

ones). They could later be copied into data structures representing some delimited continuations [43, 39], perhaps even representing and describing control [17, 46, 36]. This should also enable **introspection**, by permitting primitives inspecting the current call stack, perhaps using Ian Taylor's `libbacktrace`. Also, such an introspection might perhaps be implemented [**mitchell:2001:alp**] with two nearly twin `refpersys` processes, one of them driving a `gdb` process[24].

REFPERSYS should (like CAIA and its predecessor MALICE did [35, 37, 36]) have some expert system shell [25, 33] and meta-rules to "dynamically compile" some subset of expert system rules and knowledge bases to procedural code (e.g. with a metaprogramming approach of generating *C* code, or `libgccjit` compiled code, then `dlopen(3)`-ing that code and running it at runtime. The `manydl.c` program show that this can practically be done many dozen of thousands of times on Linux desktops).

REFPERSYS will extensively use **metaprogramming** techniques, so it **should generate code** (like CAIA do) in a transpiler approach (in C, C++, -compiled into plugins and later dynamically loaded with `dlopen(3)` - maybe also JavaScript and HTML5 if we decide to have a web user interface). REFPERSYS could also later use just-in-time compilation libraries such as `libgccjit`. The domain-specific language of REFPERSYS[25] (a declarative one, with "expert system rules") should gradually increase its expressiveness and become more and more declarative and closer to mathematical formalisms.

Most Linux distributions contain lots of useful libraries or software components for REFPERSYS long-term goals, notably machine learning open source libraries like TENSORFLOW [11] or GUDHI [12]. We might at some point also need messaging libraries like 0MQ, graphical user interfaces libraries à la QT or more probably web servicing libraries like `libonion` or WT. To decrease efforts, we don't want to rewrite such libraries inside REFPERSYS (considered as a very high level, declarative, domain-specific language). Hence, we will need in REFPERSYS to generate some glue code, like SWIG does, from some **declarative description** (probably some frames or knowledge bases) of the API of these available libraries.

REFPERSYS should at first be **orthogonally persistent**. Like BISMON [50] it will load its state (its entire garbage-collected heap) from files at startup, and will dump its state[26] into files at shutdown. These state files are textual, in JSON format, and `git`-versioned, and should be portable to other 64 bits Linux computers. A

[24] Imagine some `popen` or some `g_spawn_async` or some `Poco::Process` of some `gdb refpersys 1234` process debugging the other one of pid 1234.

[25] That domain-specific language has to be defined and implemented in a bootstrapped manner.

[26] In a manner inspired by SBCL `save-lisp-and-die` primitive, or POLYML `export` primitive, or marshalling facilities of OCAML or PYTHON `pickle` module.

manifest file describing the collection of files keeping the state is probably needed.

2.2 REFPERSYS strange development cycle

Ordinary software projects tend to follow a spiral development model [6] as shown in figure 1. But REFPERSYS' development follows a strange loop [20], since it is

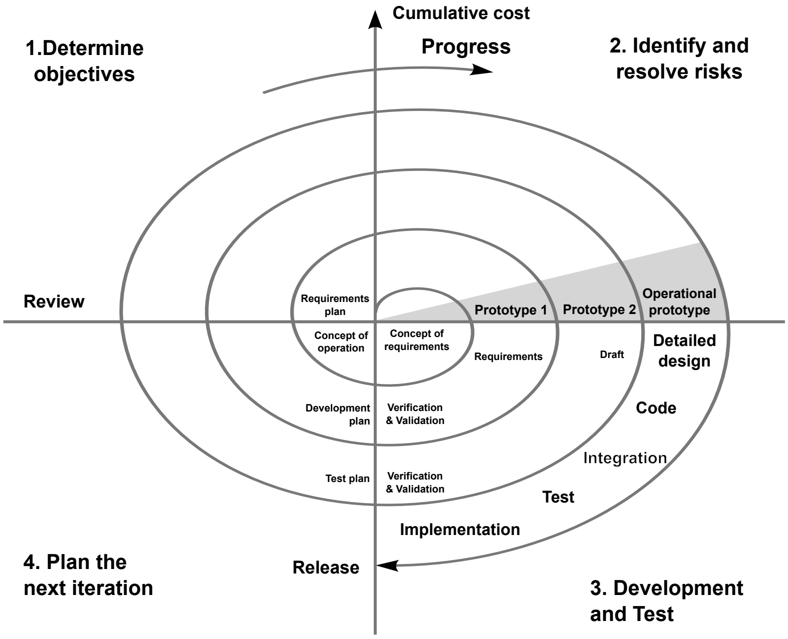
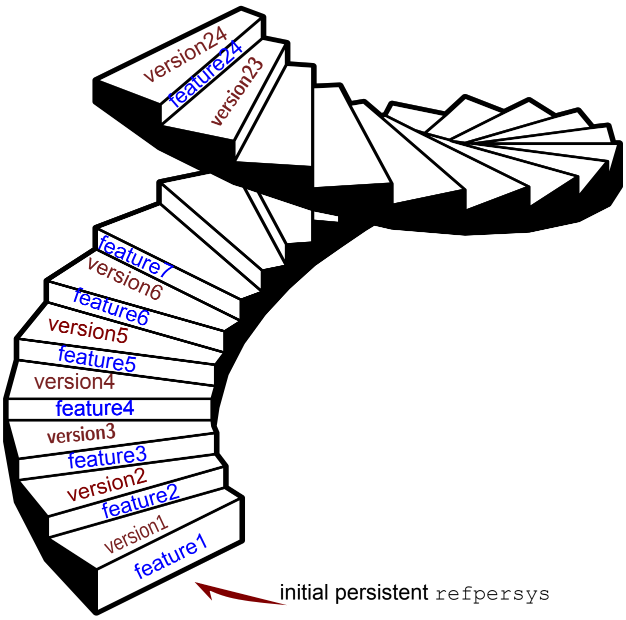


Figure 1: the **traditional spiral development** model (from Wikipedia spiral model)

bootstrapped in an evolutionary prototyping manner. It is more like a spiral staircase like in figure 2. The initial (floor) is just a persistent system, and we gradually add new code implementing more features (first entirely hand-written, later more and more parts of it replaced by REFPERSYS generated code). Of course the fun is in replacing existing hand-written code (or low-level DSL) by more expressive and generated one. So we will continuously rewrite past formalizations as a more clever and expressive ones, taking more and more advantage of REFPERSYS whole-system introspective abilities. All of EURISKO [27], CYC [28] and SELF[27] [10] (or even IO or SMALLTALK) systems and their incremental development process are inspirational.

The first significant milestone of REFPERSYS should be the ability to re-generate all its textual source files (and maybe even `git add` then `git commit` them). That

[27] SELF was even able (in hours of CPU time) to redefines its integers -even for arithmetic used inside its compiler- as bignums.



Each new feature -or small incremental change or a few of them (small `git commits`) - of REFPERSYS enables us to build and **generate** the next version of REFPERSYS, and a next feature is then added to that *improved* version, and so on repeatedly, etc....

Figure 2: the strange **REFPERSYS staircase development model** (from a figure of Spiral stairs by Lluisa Iborra from the Noun Project)

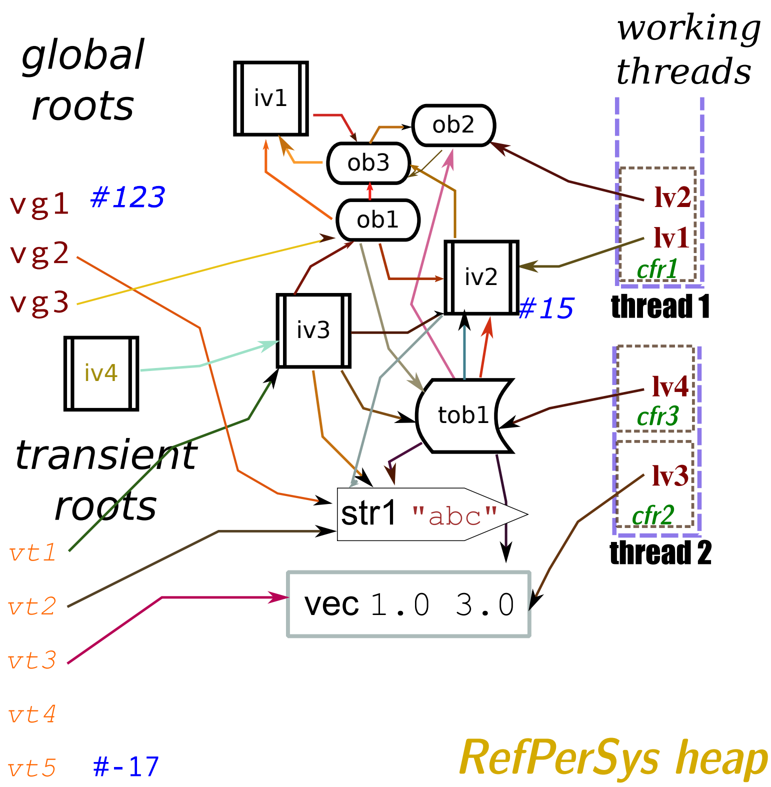
would require first implementing some simple template based machinery[28], withe the ability, like QUINE programs do, to regenerate all REFPERSYS source code (e.g. in C++, a `Makefile`, etc...). Actually REFPERSYS needs to conceptually have **self-modifying code** [51], practically implemented by systematically doing most function calls through indirect function pointers (which gets updated with `dlsym(3)`).

2.3 REFPERSYS persistent heap

When REFPERSYS is running in some multi-threaded LINUX process, the REFPERSYS persistent heap is (like Bismon's one [50]) semantically like the memory heap of most dynamic programming languages (such as PYTHON, GUILE, GO, SBCL, etc

[28] Perhaps inspired by simple designs like DJANGO tempates, but driven by frame-based REFPERSYS objects.

...). The figure 3 should give an intuition about that heap, when it is inside the virtual address space of some refpersys process. We strongly want to avoid any GIL, but multi-threaded precise efficient garbage collector implementations are quite difficult to code. However, notice that the persistence (dump as textual git-versioned disk files) of a heap uses algorithms similar to those of copying garbage collectors [53, 22].



#123	a tagged 63 bits integer
vg1	a global persisted variable
vt2	a static transient variable
ob1	a mutable persistent object
iv2	an immutable constant composite value
iv4	an immutable but **dead** constant composite value (should be GC-ed)
tob1	a transient mutable object
str1	a constant UTF-8 string value "abc"
vec	a constant vector of floats [1.0; 3.0]
lv2	a local variable inside its call frame
cfr1	a **call frame** (simplified)
thread1	a **working thread** and its call stack (simplified)

In real life, the heap may be quite large (gigabytes) and contain hundreds of global roots or transient roots, millions of objects (sometimes transient, often persistent) and many millions immutable values (some of them composite and containing values, other scalar and containing non-pointer data like strings or vectors of float do), and dozen of working threads, each having thousands of call frames with dozens of local variables each.

Figure 3: the **REFPERSYS persistent heap** (simplified)

That figure 3 shows a few global and transient roots (both being processed by the garbage collector), and several threads each having its call stack (made of call frames) with local variables in it. In that figure, if μ and μ' are two memory zones or locations (like for an object such as `ob1`, or for an immutable value `iv2`), there is an arrow $\mu \rightarrow \mu'$ if some field ϕ of μ refers to μ', that is (in C like notation) if $\boxed{\mu\text{->}\phi = \mu'}$. Different arrow colors could mean different fields ϕ, ϕ' ... etc ... The heap is actually a large directed graph and may contain cycles (e.g. $ob1 \rightarrow iv1 \rightarrow ob3 \rightarrow ob2 \rightarrow ob3$). Most values are immutable values (some of them being composite, such as `iv1`). Some immutable values are scalar (e.g. strings). Notice that `iv4` is a dead value, unreachable from others; it should be later garbage collected. Only objects have a content which may change. Since REFPERSYS is multi-threaded, the access inside every object should be thread-safe and usually is protected by a *mutex* (or *read write lock*) which is part of that object[29].

Conceptually, REFPERSYS tracing precise garbage collector should traverse the graph of references to REFPERSYS values, starting from global or transient roots and local variables inside call frames of working threads. Each REFPERSYS **value** (immutable or object) is represented by a machine word (aligned, 64 bits) which usually contains a pointer, but sometimes some tagged integer. Immutable values are often "small" (typically, less than a few dozens of words of memory, sometimes a lot more) but objects are necessarily heavier since they contain some kind of lock. closures are immutable values, containing an object representing and giving their function code (as a C function pointer inside that object), and additional closed values. In practice our garbage collector processes not only values (either immutable values or objects), but also **quasi-values** : these are a single memory zone which is allocated using the garbage collector allocation protocol, traversed by the GC when something points to it, appears inside other values (in particular, as payload of objects), but by convention should not be passed as a genuine value. So the figure 3 is a simplification.

Some values (or objects) are dead; in the figure 3, the immutable value `iv4` is not reachable from roots or local variables on the call stack of working threads. So it is dead and should eventually be reclaimed by the garbage collector.

Values -either immutable values or changeable objects- in REFPERSYS can be either **persistent** (dumped in textual state files[30], then reloaded at restart of `refpersys` process) or **transient** (that is, not dumped and not appearing in state files).

[29] Or by atomic pointers, probably the REFPERSYS class of an object is, inside it, given by some C++ field with an `std::atomic` pointer type, for efficiency reasons.

[30] In the current implementation, REFPERSYS state files should appear under `persistore/` subdirectory, and the manifest file is `rps_manifest.json` at the top directory.

The **persistence** machinery - the dump - is conceptually simple and could run in several threads: start from global roots and traverse the memory graph but ignore transient objects and transient roots and memoize previously seen persistent objects. Of course, objects should not be persisted twice, and are referred by the **object id** or **objid** in the state files produced by the dump. That *objid* is alphanumeric, randomly generated and so hopefully globally unique -like `_2om48kc3k5R02d3ktW` for example- in our current implementation; exactly like UUIDS should be. Notice the conceptual similarity between REFPERSYS dump algorithm and its tracing garbage collector: both are traversing the graph of references inside the heap.

The global roots are objects. Use the C++ functions `rps_each_root_object` to iterate on them, `rps_add_root_object` to add one, `rps_remove_root_object` to remove one, `rps_is_root_object` to test if an object is a global root, `rps_set_root_objects` to get the set of all of them, and`rps_nb_root_objects` to get their number. Of course, some global roots can be transient objects, but all of them are roots for the garbage collector.

The initial loading machinery (recreating a suitable heap - and rebuilding a graph of references inspired by figure 3, without any transient stuff) from its previous dumped state) is first creating empty all objects, then later filling each of them. However, for efficiency, we may want to load the heap in parallel, using several loader threads. This could be easy if, after having created all objects as empty, and loaded plugins (i.e. `dlopen`-ing many `*.so` files), REFPERSYS processes each state file in a potentially different loading thread.

2.4 Agenda and multi-threading in REFPERSYS

Once REFPERSYS persistence is implemented and provides some meta-programming facilities, we can define and use some agenda machinery. The insight is that REFPERSYS is running several [**butenhof:1997:programming**, 5] **worker threads**[31] known to its garbage collector (which might also need its own managing and synchronizing thread, which will mostly stay idle.). Our **agenda** is the central mechanism of REFPERSYS feeding these worker threads with some work to do, using **tasklets** representing a small amount of work to be done.

Each worker thread is indefinitely looping like this:

1. it runs occasionally some housekeeping processing, notably garbage collection work. This is where garbage collection gets synchronized. Occasionally,

[31] Concretely, this means `pthreads(7)`, perhaps wrapped as C++11 threads, QT5 threads, GLIB threads, etc ....

some new tasklets could be "auto-magically" inserted in the agenda at this point[32], e.g. to run some code when some input data is available on some file descriptor for a `pipe(7)` or a `tcp(7)` socket, or to run some code every tenth of second, or to handle graceful termination when getting a `SIGTERM`[33] `signal(7)`.

2. it waits, if so needed (probably using PTHREADS condition variables), for the agenda to become non-empty

3. it chooses a tasklet τ to run inside the agenda. That tasklet is taken, so removed from the agenda.

4. it runs that tasklet τ for a small amount of time (a few dozen of milliseconds, typically), called a **step**[34]. Of course during that step the agenda can (and usually will) change, and perhaps the same tasklet τ would be added again into the agenda, with maybe several other tasklets. Or on the contrary, running τ could remove one or several other tasklets $\tau_1, \tau_2 \ldots$ from the agenda, and add other ones $\tau'_1, \tau'_2, \ldots$ there.

5. that loop is repeated (unless REFPERSYS is stopped).

The number of worker threads is fixed and small. Typically one worker thread per processor core (so 3 on a small laptop, 20 or 30 on a big desktop). Of course the agenda mechanism requires synchronization through locks or mutexes and PTHREAD condition variables [5].

In addition of the worker thread, some additional slave threads could be needed, in particular to handle some event loop (and serve HTTP requests). Of course the running steps should appropriately lock objects, to avoid aftermath and synchronize properly their mutation.

The concrete organization of the REFPERSYS agenda has to be precisely defined. It could be, as BISMON has, a small data structure made of several first-in first-out queues, e.g. a queue of high priority tasklets, another of medium priority tasklets, one of low priority tasklets, etc..., with the agenda mechanism choosing in the non-empty queue of highest priority its tasklet staying in front.

[32] Or such tasklets could be very carefully added into the agenda from non-worker threads organized in a producer-consumer fashion -such as those started by `libonion`-, respecting our GC invariants. This is a delicate issue !

[33] Read also `signal-safety(7)` and consider using `signalfd(2)` or pipe-to-self tricks inspired by QT approach to UNIX signal handling. Notice that `timerfd_create(2)` might also be useful for tasklets to be added periodically in some event loop around `poll(2)`.

[34] Calling blocking system calls such as `poll(2)` or `read(2)` from a pipe or socket should be forbidden here, because a step should run quickly, in milliseconds.

2.5 Metaprogramming and introspection in REFPERSYS

Metaprogramming is defined in Wikipedia as "a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running". That design idea is central to many Artificial Intelligence systems and AI inspired languages[35] and is also common in software engineering[36] [27, 26, 28, 37, 35, 36, 38, 40, 39, 46, 48, 47, 49, 50, 51, 2, 7, 10, 13, 15, 17, 18, 19, 20, 23, 24, 25, 31, 32, 33, 41, 45, 44]. Generating some "source" code at build time is usual practice, advocated also by the NINJA build system, and theorized (around 1930, before even computers existed) in the CHURCH-TURING thesis. Related concepts include the famous (but undecidable) **halting problem** (whose proof involves a metaprogramming approach [21]), hygienic macros, and Rice's theorem.

Practically speaking [2], metaprogramming is easier achieved by explicitly representing (maybe incomplete) code with abstract syntax trees (or AST), maybe with some holes for metavariables for their later explicit substitution, in the spirit of DJANGO templates or of COMMON LISP macros or SCHEME macros. A practical way to implement such a template machinery for generating C or C++ code is given by GCC MELT code chunks [48, 47, 45, 49], where a piece of C (or C++) code with holes (or metavariables) `$hellochunk` and `$msg` is given through the "macro-string" `#{/*$hellochunk#_here*/ printf("hello %s\n", $msg);}#` ...
Later, such a macro-string or code chunk can be expanded by filling the holes, that is expanding the metavariables (e.g.`$msg`) appropriately. Such an expansion might be recursive, since some hole filling (or metavariable replacement) could in turn trigger expansions of other macro-strings. In practice, REFPERSYS will use similar code chunks and macro-expansion to generate its C (or C++) code, and some initial ad-hoc integrated development environment (or IDE) will have to be coded, handling passively some persistent store. The expansion will be done through some scripting language (or *domain specific language*, a.k.a. DSL) which has to be implemented inside our IDE.

Metaprogramming involves code generation (using source-to-source ahead-of-time and/or just-in-time[37] compilation techniques [**Aho:2006:dragon-book**], and in REFPERSYS is useful for many tasks, such as generating the garbage collection

[35] See also SCHEME 48, SBCL, RUST, even C++ *templates*, CHICKEN SCHEME, METAOCAML, the ECLIPSE Constraint Programming System, RASCAL, NEMERLE, COCCINELLE, OCSIGEN, GNU PROLOG, CLIPS, GPP, SWIG, ANTLR, IBURG, Gnu BISON, etc ...

[36] A typical example is the GCC compiler, or AUTOCONF, and transpiler approaches

[37] Several JIT compilation libraries exist, notably `libgccjit` provided inside recent GCC compilers.

support routines for scanning or forwarding, and the loading and dumping routines needed for persistence (in the spirit of RPCGEN, SWIG and other serialization frameworks).

In REFPERSYS, metaprogramming is often and practically achieved (like in [48, 50, 37, 36] and our `manydl.c` example program), by generating some C or C++ code in a temporary file[38] like `/tmp/rpsgen123.c`, compiling that file [16] into a generated plugin `/tmp/rpsgen123.so` by running a process such as `gcc -fPIC -Wall -O -g -shared /tmp/rpsgen123.c -lsomething -o /tmp/rpsgen123.so` and waiting for its successful completion, then `dlopen(3)`-ing that newly generated `/tmp/rpsgen123.so`, in a manner compatible with our garbage collection and agenda invariants. We might later care about carefully `dlclose(3)`-ing that generated plugin, but in practice we accept some limited virtual memory plugin leak, and we could just dump appropriately our persistent state by mentioning in some generated Manifest file those plugins which should be saved (as generated C code) with the state.

Reflection is "the ability of a process to examine, introspect, and modify its own structure and behavior" and also, for self-reflection, the capacity " to exercise introspection and to attempt to learn more about their fundamental nature and essence". (Wikipedia). It is advocated (in [36]) that a similar approach is (painfully) achievable in AI systems, and it would need both clever backtracking and backtracing techniques. Libraries such as Ian Taylor's `libbacktrace` (which wants most of the code to be compiled with DWARF debugging information[39]) are helpful.

Our precise garbage collector (see §3 below and [**rafkind:2009:precise-gc**], or QISH) wants local variables holding garbage collected pointers to be known to the GC. In practice, the REFPERSYS call frame is some explicit local `struct` named _ in generated C code[40]. Such explicited local frames can often be optimized by `GCC` or `g++` (invoked with `-O2`).

As suggested by Pitrat (see [37, 35, 36]), call stack reflection and backtrace is the elementary brick of more sophisticated *introspection* techniques. At some point, our REFPERSYS system should inspect its call stack and may take decisions after that. A typical approach would be to run such introspection once in a while (e.g. every 0.1 second on the average[41], in the inference engine of some expert system or knowledge

[38]There are practical reasons to generate these temporary files outside of `/tmp/`, which gets cleaned at reboot.

[39]In practice we should compile our or other C or C++ code with both `-O2 -g` passed while invoking `GCC` or `g++`, and this is indeed possible and practically works well enough.

[40]Like Bismon does, see its `LOCAL_BM` macro. See also the `CAMLparam`*i* and `CAMLlocal`*j* C macros of OCAML, and the `Py_VISIT` and `Py_DECREF` and other macros of PYTHON, the *foreign function interface* of SBCL, etc ...

[41]Timing considerations are essential, practically speaking, in REFPERSYS. See `time(7)` man

base component of REFPERSYS.

Since we aim to be able to re-generate most (and hopefully all) of REFPERSYS code (in C or in C++), having simple **coding conventions** does matter: every REFPERSYS-defined C or C++ identifier should start with `rps_` in lower, upper, or mixed case (e.g. also `RPS_` or `Rps_`). Every C or C++ function, even `static inline` ones appearing in header files, has its name starting with `rps_` and is *globally* unique to the entire `refpersys` program. The C (or C++) code should be automatically indented[42] using Gnu INDENT or ASTYLE. Every named `struct` (in C) should have its tag matching `rps_*st`. Every `typedef`-ed data type should have its name matching `rps_*t`. Every named `enum` should have its tag matching `rps_*en` and the various enumerated values like `RPS_*`. Even in cases the C (or the C++) language allows several name spaces[43], we don't use that facility. Hence we refuse to code the common `typedef struct rpsfoo_t rpsfoo_t;` but prefer instead (inspired by GTK) coding `typedef struct rps_foo_st rps_foo_t`. Of course, names of local variables (that is automatic variables with their lexical scope limited to some small C or C++ block) could be as short as a single letter such as `i`. In general, our C or C++ code is written with the hope of being easily able to regenerate it.

3 The data and object models of REFPERSYS

The data is what is processed by REFPERSYS, and is made of values (and, internally for the GC, also of quasi-values, which are pointers to GC-managed memory zones). The object model is defining our classes, our single inheritance mechanism, our message sending protocol (see §3.8.2).

3.1 how data should be processed in REFPERSYS

REFPERSYS aiming to be first a good old fashioned AI system (GOFAI), better known as symbolic artificial intelligence system, it is targetting mostly symbolic computation, in particular using a semantic network or other forms of mathematical finite but large graph representations, in particular abstract syntax trees[44] of generated programs, of internal rules or expressions, by some internal metaprogramming

page.

[42]With the social convention that REFPERSYS contributors are running `omake indent` or `make indent` before every `git commit`!

[43]In C, having both a type and a label named `foo` is permitted, but we refuse such non-sense.

[44]Practically speaking, abstract syntax trees are in fact at least finite directed oriented graphs and could even have cycles if you relate a symbol to its properties.

machinery. So REFPERSYS objects should have a finite but changing set of attributes or properties and be organized, as in most object-oriented languages. Hence, documents, hypertext, high-level source code, ontologies, knowledge bases, expert systems, implementation of some inference engine guided by metarules, etc ... should all be easily and conveniently representable[45] and processable, as some evolving subgraph of REFPERSYS values.

Since REFPERSYS objects are the only mutable values, they keep not only their synchronization data, but also attributes or properties, components, and some extra **payload**[46]. See also §3.4 below.

The REFPERSYS worker threads, organized in a small thread pool[47] are somehow organized in some **agenda**[48] mechanism. Informally, the agenda is a clever organization (perhaps a few mostly FIFO queue of elementary tasklets, or something more complex). Each such tasklet runs for a short time[49] and may, while running, update that agenda by adding further runnable tasklets to it, or by removing some of them. The agenda itself should be somehow reified and partly persistent, and tasklets are REFPERSYS objects. Of course some tasklets (e.g. those directly related to the user interface, e.g. AJAX or QT callbacks) are transient.

3.2 data at the low and high levels

REFPERSYS mostly handle values[50], which can be either "light" immutable values or "heavy" mutable objects. Our data model is inspired by the OBJVLISP model (or CLOS, see also the Common Lisp HyperSpec) common in most Lisp implementations [41, 13, 7] and inspired by SMALLTALK [23]. Also, a value can be transient or persistent. Each REFPERSYS value fits in one 64 bits machine word[51], so is nicely represented as an aligned pointer (ending with a 0 bit) or a tagged integer (63 bits, with the least significant bit being set to 1). Values are usually pointers to complex structures, so, per the x86-64 calling conventions, are word aligned (address multiple of 8 bytes). Let's call *genuine values* those that are not `null` and not tagged pointers (so either immutable values or objects in figure 3). These **genuine values** (and also

[45] So artifacts like XML documents, HTML5 or XHTML hypertexts, JSON data, YAML representations should all be easily representable and inspirational for REFPERSYS data and its processing.

[46] From the GC point of view, payloads are quasi-values ...

[47] Threads are heavy resources, each of them needing a call stack and, practically speaking, a processor core to run. We surely want to have at most a dozen of worker threads.

[48] In Latin, "agenda" means "things which have to be done or completed".

[49] In practice, several dozens of milliseconds, to play nice with human interaction and be friendly with our garbage collector

[50] In particular, only CLOSURES are applied, to arguments which are values (read more about λ-calculus); or messages are sent, to values with perhaps additional value arguments. Internally, our GC also handle quasi-values.

[51] Remember: REFPERSYS targets only Linux x86-64 systems!

quasi-values) are practically implemented as a tagged union[52] and each of them start with a field (probably 16 bits) identifying their concrete type.

3.2.1 values and quasi-values

The REFPERSYS garbage collector manages both values and quasi-values (that is, a single non-empty sequence of memory words, used for some garbage collected data, e.g. inside objects). But only persistent values are dumped and reloaded in the persistent store. The values which are not dumped -so not reloaded on the next run- are called *transient* values.

For pragmatical reasons, our values[53] should be both ordered and hashed, since many data structures [14], specified as some abstract data type, either uses some ordering (e.g. in red-black trees) or some hash-code (e.g. various kinds of hash tables). Because of the weird and counter-intuitive semantics of floating point numbers, the NAN should be handled specifically (it is unordered), if we box IEEE `double`s.

3.2.2 implementation details

REFPERSYS takes advantage of some practical features[54] of C on Linux x86-64:

- Practically, machine data pointers should be at least 64 bits (8 bytes) aligned[55] for large enough memory zones (i.e. most practical `struct`-s), annd preferably 128 bits, that is 16 bytes, aligned. See also the `alignof` macro of `<stdalign.h>` and the `aligned` type attribute.
- Limited type-punning abilities. Assume we have two `struct`-ures definitions, so `struct s1` and `struct s2`. Assume that both `s1` and `s2` *start* with the same *common* fields `unsigned num;` then `void*ptr;` followed by `char str[24];`. Assume that a pointer `p` points to a valid memory zone, whose alignment (respectively size) are at least all of `alignof(struct s1)`, `sizeof(struct s1)`, `alignof(struct s2)`, `sizeof(struct s2)`: so we have `alignof(typeof(*p)) >= alignof(struct s1) && sizeof(*p) >= sizeof(struct s1)` and `alignof(typeof(*p)) >= alignof(struct s2) && sizeof(*p) >=`

[52] An old example of tagged unions in C is the X11 event structure, but GUILE and OCAML use similar implementation tricks.

[53] Of course, quasi-values need not to be ordered and hashed!

[54] We don't really care if these features are not exactly standard C11 [**c11-standard:2011**], because we strongly believe they are present on practical Linux x86-64 computers.

[55] The X86-64 or AMD64 instruction set architecture allows in principle unaligned memory accesses, but these are very slow and unfriendly to cache coherence hardware implementations.

sizeof(struct s2). Then: ((struct s1*)p)->num and ((struct s2*)p)->num both refer to the same memory location and number there; ((struct s1*)p)->ptr and ((struct s2*)p)->ptr both refer to the same memory location and pointer there; and of course ((struct s1*)p)->str and ((struct s2*)p)->str is the *same* string. See also may_alias, warn_if_not_aligned, aligned, transparent_union GCC type attributes, the -fms-extensions option to GCC, and its unnamed fields ability.

- Tail call optimization, practically provided in *some* common cases by recent GCC or CLANG/LLVM compilers (requiring probably -O2 compiler flag).
- Common extensions to the C language, notably statement exprs (very useful), label as values (or "computed goto"-s), typeof, zero-length arrays and flexible array members, return addresses and other built-ins, may be used in REFPERSYS code.

Practically speaking, every REFPERSYS value or quasi-value (see our Rps_QuasiZone class) which sits in memory[56] is represented in some class inherited from Rps_ZoneValue For instance, our string values have their memory zone type declared as Rps_String, but we use the Rps_StringValue class to construct them. In Rps_String the field _sbuf is a flexible array member, and by convention contains _bytsiz + 1 bytes (terminated with a 0 byte), is validly UTF-8 encoded, aligned to 4 bytes and nul-byte terminated. Hash codes cannot be 0 and are lazily computed (so the rps_strhash field is computed once when it was 0). The Rps_Type::String is some enumerator inside a global enum. Strings are ordered naturally, using strcmp on their rps_strdata bytes.

The refpersys executable is handling files either from the REFPERSYS home directory (obtained inside C++ code using a rps_homedir() call), given by $REFPERSYS_HOME or $HOME environment variables or thru the -refpersys-home program argument, or from the REFPERSYS load directory (by default the source directory, or given thru the -load program argument). User preferences should go into the REFPERSYS home directory, e.g. as the .refpersys.json file there.

@@TODO: should explain more implementation details in C++ terms?

[56]This excludes tagged integers, and that memory zone is at least word aligned to 8 bytes.

3.3 immutable values

By definition, immutable values don't change. All their useful bits[57] stay unchanged as long as the value is alive. Some values are scalar (strings, vectors of floats, perhaps bitmaps[58] if we reify them). Other values are composite.

Since objects are fundamental, we want to keep finite collections of them. In particular, REFPERSYS will reify (represent as first-class *immutable* values) **tuples** of objects and finite **sets** of objects as values, and also , and they are the common composite values of REFPERSYS. A tuple is obviously represented by boxing a sequence of object references (i.e. pointers). A set would be represented by an ordered sequence of object pointers, with membership efficiently testable by a $O(log\ n)$ time binary search algorithm). We expect most of tuples and sets to be small and fitting in an L1 or L2 cache line, so their processing should be efficient.

In REFPERSYS, closures -that is first-class procedural values, like in SCHEME[59] [24, 31, 40, 39, 2], HOP or BIGLOO [44], COMMON LISP, JAVASCRIPT - are also immutable values. The closed values -binding free variables of the closure- inside such closures are arbitrary, but fixed, and won't change during the lifetime of that closure. The function code inside them is given by some fixed object reifying that code, and probably useful to generate the "source" code (e.g. as generated *C* or *C++* code) of that function. The mangled name, later "`dlsym(3)`-ed", of that function in its ELF `*.so` shared object file [16, 30] is somehow related[60] to the *objid* of that object. Closures are absolutely essential in REFPERSYS, since they are the only way to refer to executable machine code. Even method implementations are using closures, since the virtual method table in REFPERSYS classes (actually, their payload) is referring to closures (is is an association between selectors (like in Objective-C, but reified as objects) and closures implementing methods, à la OBJVLISP [13, 1]).

We may consider also having in REFPERSYS immutable node[61] instances: like mutable objects (see §3.4 below), each of them would have a class (with single-inheritance), attributes and components. But since they are immutable, they have no objid, no locking mechanism, and their class, attributes and components would be fixed and defined at their creation time.

In C++ code, values are `Rps_Value`, a "smart" value container, actually a single

[57] For housekeeping purposes, our garbage collector may reserve a few bits, e.g. for tri-color marking [53]

[58] In practice, bitmaps or pixmaps would rather be the payload of some objects, see below.

[59] Try for example GNU GUILE following this tutorial.

[60] For a fictional example, an object of objid `_61NdgIkKhwD04la094` might be related to some ELF function of name close to `rps_61NdgIkKhwD04la094`, etc ...

[61] REFPERSYSnodes are generalizing BISMON nodes [50], which are immutable, have an object connective and a sequence of sons which are arbitrary values. Perhaps "node" is a wrong word and we could name them "records" or "structures".

word. That class is specialized into helper subclasses, such as `Rps_StringValue`, `Rps_DoubleValue` etc.

3.3.1 immutable scalar values

Immutable scalar values include:

- **strings**, persisted as JSON strings; their internal representation is `Rps_String` using an UTF-8 encoding. In C++, use `Rps_String::make` to make one, or construct an `Rps_StringValue` with a C or C++ UTF-8 encoded string, a `std::string`, or a `QString`.
- boxed **doubles**[62] (which cannot hold an IEEE-754 `NaN`, which is incomparable), persisted as JSON doubles; their internal boxed representation is `Rps_Double`. In C++, use `Rps_Double::make` to make one, or construct an `Rps_DoubleValue` with a C `double`.

3.3.2 immutable composite values

Immutable composite values include:

- **tuples** of object references. Their memory representation is a `Rps_TupleOb` zone, and it has both `Rps_TupleOb::make` and `Rps_TupleOb::collect` static functions. But use `Rps_TupleValue` to build them. @@TODO: should explain more
- **set** of object references; Their memory representation is a `Rps_SetOb` zone, and it has both `Rps_SetOb::make` and `Rps_SetOb::collect` static functions. But use `Rps_SetValue` to build them. @@TODO: should explain more

@@TODO: should explain a lot more

3.4 mutable objects

Practically speaking, mutable objects are heavy, since they should carry inside them locking devices[63] for multi-threading support. And each object carries an association between attributes (playing the role of arbitrary keys) and their corresponding value.

[62] See `floating-point-gui.de` for more about IEEE 754 double precision numbers on current computers. This is actually a difficult topic.

[63] At the implementation level, think of some mutex or preferably some read-write lock, so `pthread_mutex_init` or `pthread_rwlock_init` or C++11 equivalents.

In addition, an object can carry its payload[64] for stuff which does not fit into that model. For example, an object may carry as its payload a dictionary associating machine strings to values, or an hash-table of triplets, or an opened `FILE*` handle, or file descriptor[65], or some TENSORFLOW or GHUDI data for machine learning purposes, maybe something related to `libonion` or `ZeroMQ`, some GMPLIB big number, some `PPL` polyhedra, maybe some QT graphical widget, etc .... Of course every object has its class (which is itself an object, having a metaclass) and carries a function pointer[66], for REFPERSYS closures.

Notice that a generational GC approach moving data is possible for some immutable values, but not for objects and their optional payload, since REFPERSYS objects contain locks and payloads that are dealt with through external functions requiring fixed, unchangeable, pointers.

In C++ code, object references are `Rps_ObjectRef`, a "smart" object container, actually a single word. They are persisted by their *objid* in JSON format as a string.

3.4.1 objects as frame-like data

REFPERSYS objects are quite flexible, even more than JAVASCRIPT[67] ones; they take some inspiration from RLL [18], EURISKO [27, 26], CYC and its CYCL [28], and more recently the *Semantic Web* and its OWL. An object has attributes (usually a few ones, but perhaps many of them) and components (again, perhaps 0 or a few of them, but in rares cases thousands of them), and some optional payload (quite often, it is missing). Notice the similarity with JAVASCRIPT objects (which calls *fields* with *keys* what REFPERSYS calls attributes, and array elements what are REFPERSYS components). However, JAVASCRIPT is (like IO or SELF [10]) a prototype-based object programming language, while REFPERSYS remains, like JAVA or C#, or COMMON LISP (and of course C++ or GO), a more or less class-based object programming language with single inheritance: in REFPERSYS (like in C# or JAVA) all objects are indirect instances of the same single top-level class (which is reified as an object, using some metaclass machinery).

Through their flexible attributes (each of them can be fetched, added, removed

[64] Every payload belongs to a single object, its owner!

[65] Finalizers are practically not enough to handle these, even if they are useful, in our GC, as a last resort mesure!

[66] That function pointer should be, for efficiency reasons (we don't want to lock an object to get that pointer!) `atomic` in C parlance, and might be set using `dlsym(3)`.

[67] Remember that in JAVASCRIPT **`ob.fl`** is defined to be the same as `ob['fl']`, the equivalent of fetching attribute or field `fl` from an object `ob`. And **`ob[1]`** is like component of index 1 in object `ob`.

or changed during the lifetime of the containing object), REFPERSYS objects[68] can be used to represent *frames* (read about frame languages) or semantic networks and other forms of graph databases (held entirely in memory).

REFPERSYS objects keep (like BISMON objects do [50]) their modification time (or *objmtime*) and that timestamp[69] may be useful to decide some further processing (à la `make`).

@@TODO: should explain a lot more

3.4.2 concrete examples of objects

In the below examples, ρ *"some foo"* would be the "reification" of *some foo*, that is a REFPERSYS value for that *some foo*. When we are certain that *some foo* is represented by a REFPERSYS mutable *object* (not some immutable value), we would write Ω *"some foo"* instead of ρ *"some foo"* ....

REFPERSYS contributors should be known to the REFPERSYS system[70]. So a typical contributor would be "reified" by an object of objid `_3a9otsskmcJ04v9S7n` representing him, shown in figure 4.

$\in$	Ω " contributor class"	the class
Ω "first name" :	string `"Basile"`	attribute
Ω "last name" :	string `"Starynkevitch"`	attribute
Ω "email" :	string `"basile@starynkevitch.net"`	attribute
Ω "year of birth" :	tagged integer `#1959`	attribute
Ω "friends" :	set `{ _6eO8kozzUh801dHVt7 _7zpWFJ2npj001ZfVvq }`	attribute

Ω "Basile STARYNKEVITCH" $\equiv$ `_3a9otsskmcJ04v9S7n`

Figure 4: An example of object representing a contributor

The REFPERSYS system could flexibly add additional information, e.g. with an attribute Ω *"authored"* associated to some tuples of objects authored by that Ω "Basile STARYNKEVITCH". Should his email change, that could be represented by overwriting attribute Ω "email" with a string such as `"basile@refpersys.org"`. It is important that attributes are themselves objects: one could imagine that the

[68] REFPERSYS objects are also -conceptually- inspired by the GOBJECT framework of GNOME.

[69] The timestamp is implemented as a `double` floating point representing elapsed seconds since the UNIX Epoch, obtained with `clock_gettime(2)` using `CLOCK_REALTIME`.

[70] In 2019, we ignore subtle issues like European GDPR since it focuses on *transmission* of personal data, assuming that every contributor to REFPERSYS consciously decided to contribute to REFPERSYS on his own free will. We believe, similarly, that `git` users also have decided to use it with their freedom. We are not lawyers.

object Ω "email" could contain an attribute Ω "how to display" associated with a closure, which would be applied to show that email cleverly (e.g. as some `<a href='mailto:...'>` HTML5 element).

Another example of object might be some code chunk to safely print into `$fil` an integer `$i`, e.g. `#{ if ($fil != NULL) fprintf($fil, "%d", $i);}#` might be represented as in figure 5:

∈	Ω "code chunk class"	the class
Ω "metavariables" :	set { Ω "`$i`", Ω "`$fil`" }	attribute
Ω "target language" :	Ω "*C* language"	attribute
[0]	string `" if ("`	component
[1]	object Ω "`$fil`"	component
[2]	string `" != NULL) fprintf("`	component
[3]	object Ω "`$fil`"	component
[4]	string `", \"%d\", "`	component
[5]	object Ω "`$i`"	component
[6]	string `");"`	component

Figure 5: a simplified example of code chunk

REFPERSYSis coded **only** for LINUX/X86-64 with an English locale, using UTF-8.

3.4.3 object payloads

Objects may have some *optional* unique **payload**. The payload is owned by a single object (its owner). The payload data is generally mutable, and contains stuff which does not fit into the object model (see §3.8 below). The payload may be persistent, but some payloads are obviously transient (e.g. an opened file handle, or a Qt widget). At the implementation level, a payload is some garbage-collected quasi-value whose "type" starts with `Payl` in C++. By convention, the class of an object may require a particular payload: for example objects which are classes have to have a payload which is a class information.

Payload may be:

- a class information (inside classes, `Rps_Type::PaylClassInfo` and `Rps_PayloadClassInfo`) which gives the superclass object and the dictionnary of methods (see §3.8.2 and figure 12).
- a mutable set of objects `Rps_Type::PaylSetOb` (see figure 13).
- a mutable vector of objects `Rps_Type::PaylVectOb` (see figure **??**).

- a mutable vector of values `Rps_Type::PaylVectVal`
- a mutable association from objects to values `Rps_Type::PaylAssoc`
- a mutable binary relation between objects `Rps_Type::PaylRelation`
- a mutable string buffer `Rps_Type::PaylStrBuf`
- etc ...

Some payloads could be erased or replaced by another kind of payload, but some payloads are not erasable; for example, it makes no sense to replace a class information payload by a string buffer one in the same class object. Hence payloads have a `is_erasable` member function in C++.

3.5 File naming

Our ***C++17* hand-written files** are named like `*_rps.cc` for source files, and `*_rps.hh` for header files, with a special case for the common `refpersys.hh` super-header file including most others. If they use QT5 extensions requiring its `moc` they would be named `*qrps.cc` for Qt C++ source files and `*qrps.hh` Qt C++ header files.

Documentation goes under `doc/` (preferably in LaTeX, probably the LUALATEX variant). It could need `inkscape` version 0.92 or better and GRAPHVIZ version 2.40 or better.

Temporary C++ generated files, including those generated by `moc` should be named with something starting with an underscore _ if they don't need to be `git commit`-ed.

Permanent C++ generated files which have to be version controlled so `git add`-ed go under `refpersys/generated/` directory.

Every hand written C++ file should have a proper GPLv3+ comment at start. The copyright owner is the REFPERSYS team. We mention `refpersys.org`, and we list every human member of it.

3.6 Building `refpersys` executable

Dependencies: We need the latest JsonCpp library, at least its version `1.7`. We need QT5, at least version `5.12`. Our build automation system is `omake` version `0.10.3` or better. We depend upon GNU `bash` installed as `/bin/bash`. We need also `pkg-config` version 0.29 or better, suitably configured to play nice with at least QT5. We could later need GNU `unistring` (for UTF-8 processing) and maybe ANTLR4 (as a parser generator).

3.7 REFPERSYS workflow

As long as we are very few and part-time on that REFPERSYS project, we essentially use `git` as an improved *centralized* version control system à la `svn` (so the *distributed* nature of `git` is irrelevant for us in 2019. Itc could become important when the REFPERSYS matures and generates a lot of files). By social convention: we `git commit` often (e.g. every hour or two of work). Before that, we `omake indent` and we ensure that the code is buildable with `omake clean` followed by `omake -project` before any `git commit`. We format and indent manually written C++ code with `omake indent` or using our `indent-cxx-files.sh` shell script before any `git push`.

Our `git commit` messages given by `git log` are starting with a short sentence in English (ASCII characters only). If more than one sentence is needed, the following ones should start with a blank line.

Glossary

Artificial General Intelligence Artificial general intelligence (**AGI**) refers to the specific capacity of a machine to learn and understand any intellectual task that can be performed by a human being. It is the primary goal of some artificial intelligence research, and is sometimes referred to as "strong AI" or "full AI" . 1

metaknowledge Metaknowedge is knowledge about knowledge, and is an inclusive term spanning several disciplines. Bibliography, the academic study of books, and epistemology, the philosophical study of knowledge, are examples of metaknowledge. Even the tagging of documents could be considered as metaknowledge. In AGI, metaknowledge refers to the knowledge about knowledge-based systems. A declarative system might be guided by metarules, that is "expert system" rules to compile or interpret other rules (maybe themselves). . 2

Reflection Reflection is (according to *Wikipedia*) "the ability of a process to examine, introspect, and modify its own structure and behavior", and related to Self-reflection, the capacity of humans (and hopefully of artificial cognitive systems, like REFPERSYS should become) "to exercise introspection and to attempt to learn more about their fundamental nature and essence". . 2

In practice, such a code chunk representation could be more compact; we could assume that both the **`if`** keyword of C and the `fprintf` and NULL C identifiers occur frequently enough to be refactored and each reified into its own object. Of course, it might make sense to add an Ω "author" attribute in our code chunk, whose value would be our `_3a9otsskmcJ04v9S7n` object of figure 4 above.

Obviously, some kind of data don't fit exactly into such simple objects. Some objects might represent a big hashtable of triplets, and such data has to be the payload of that object. Other objects might reify sorted dictionaries mapping strings to values, and their payload could be some red-black trees whose internal nodes would be GC-managed quasi-values. And an opened `FILE*` would be represented and reified as some object carrying a payload with some `FILE* rps_filehandle;` field.

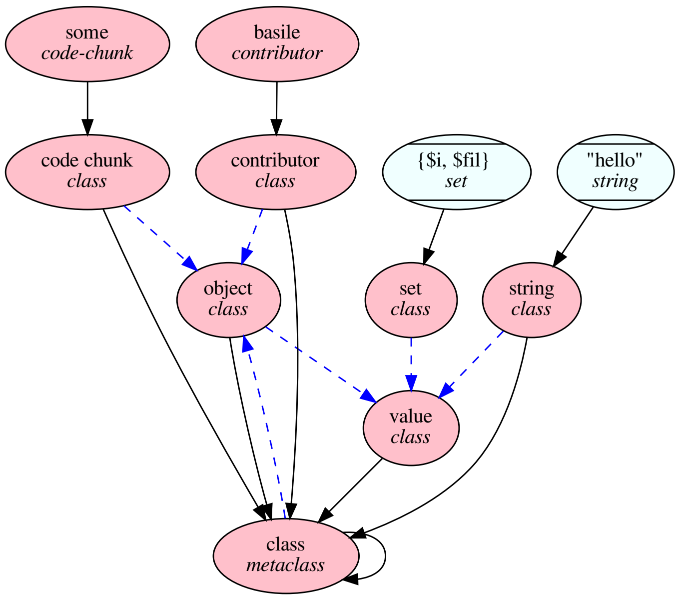
@@TODO: TO BE WRITTEN

3.8 the REFPERSYS object model

Every REFPERSYS value belongs to some single class, reified as a particular REFPERSYS object. We first explain the inheritance graph (see §3.8.1), and then the message sending protocol (see §3.8.2).

3.8.1 REFPERSYS inheritance graph

Every non-nil REFPERSYS value belongs to a single class, defining its behavior by the set of selectors it is understanding for message sending. The figure 6 shows (with simplification) the single-inheritance graph of REFPERSYS values. In practice we expect many hundreds of classes and at least many hundred thousands values in a mature persistent store.



- objects (pink background)
- immutable values (azure background, horizontal lines)
- $v \longrightarrow \omega$ (straight black arrow) means : v is instance of class ω
- $\omega_1 \dashrightarrow \omega_2$ (dashed blue arrow) means : class ω_1 is subclass of ω_2

Figure 6: The simplified inheritance graph in REFPERSYS

That figure 6 shows several objects:

- the `OBJECT` class, superclass of every object;
- the `CLASS` metaclass, class of every class;
- the `VALUE` class, ultimate class of every value;

- the `CODE CHUNK` class, for code chunks like in figure 5;
- the `CONTRIBUTOR` class, for reified contributors like in figure 4;
- the `STRING` class, of immutable string values;
- the `SET` class, of immutable set of objects;
- the `BASILE` contributor object of figure 4;
- some *code-chunk* object, like in figure 5;

and several immutable values:

- the `"hello"` string;
- set of two chunk metavariables { Ω(`$i`), Ω(`$fil`) } which appears as value of attribute Ω "metavariables" in figure 5

A REFPERSYS class object should contain, in its payload some *class information*:

- the sequence of its direct then indirect super classes
- a flexible dispatch table or virtual method table[71] associating selectors to closures handling messages with them. We call that association the direct *method dictionary* of that class. It should be implemented efficiently (perhaps using caching techniques local to each occurrence of sending).

Both are changeable. Any object can change its class at will at any time. A class can have new methods added or removed at any time. A class can change its superclass at will[72].

3.8.2 REFPERSYS message sending

Our message sending protocol is inspired by those of SMALLTALK, COMMON LISP, OCAML, JAVA. Every message send has a receiver ρ (the target of the message sending), a selector - some object ω_{sel} (what do we send) and optional extra arguments $\alpha_1 \dots \alpha_n$ (so $n = 0$ when we don't have extra arguments). Conceptually, what is happening is some loop:

[71] Our methods are *always* virtual, like for SMALLTALK; Conceptually REFPERSYS don't have any non virtual methods.

[72] However, this should be done with care, avoiding additional circularities in the inheritance graph.

- let κ be initially the class of ρ, the receiver or target.
- look into the method dictionary δ of class κ; if the selector ω_{sel} is associated to method μ (some closure), apply that μ to ρ, $\alpha_1, \dots \alpha_n$; the result of this application is the result of the message sending.
- if κ is the topmost VALUE class, the message sending has failed.
- otherwise, replace κ by its direct superclass κ' and repeat the method lookup (second step here).

In practice, we might try to use caching techniques (but later) à la SELF or JAVASCRIPT implementation to accelerate message sending. We should define what happens when no method is found (perhaps using some MESSAGE-NOT-UNDERSTOOD built-in selector à la SMALLTALK [23]), taking $\omega_{sel}, \alpha_1, \dots \alpha_n$ as arguments, or triggering some exception machinery.

4 Persistence in REFPERSYS

The persistence of REFPERSYS is an essential feature. The `refpersys` program starts by loading its persistent state (from various textual files under `persistore/` directory[73]). In the usual case, a `refpersys` process dumps its persistent state before exiting.

A manifest file named `rps_manifest.json` is describing the entire persisted state and referencing indirectly other files. The figure 7 is giving the syntax of that file.

manifest	$\leftarrow$ **{**	
	`"format": "RefPerSysFormat2019A"`	mandatory format id
	`"spaceset": [` $id_{space} \dots$ `]`	oids of spaces
	`"globalroots": [` $id_{root} \dots$ `]`	oids of global roots
	`"plugins": [` $id_{plugin} \dots$ `]`	oids of `dlopen`-ed plugins
	}	

Figure 7: syntax of the manifest file `rps_manifest.json`

If `_8J6vNYtP5E800eCr5q` is a space oid id_{space}, then the persistent space data is in JSON file `persistore/sp_8J6vNYtP5E800eCr5q-rps.json`.

[73] Of course, some other directory can be given through explicit program arguments to the `refpersys` executable.

For plugins, if `_7GIB3ma21I200tfqDs` is a plugin oid id_{plugin}, its generated C++ source code should go into the file `generated/rps_7GIB3ma21I200tfqDs-mod.cc` and the corresponding dlopen-ed plugin in `plugins/rps_7GIB3ma21I200tfqDs-mod.so` ELF shared object file.

The loader will `rps_add_root_object` every root object of given id_{root}.

@@TODO: improve

4.1 The textual data format of REFPERSYS

Each space file of id_{space} starts with a prologue whose syntax is in figure

space-prologue	← **{**	
	`"format":` `"RefPerSysFormat2019A"`	mandatory format id
	`"spaceid":` id_{space}	the id of the current space
	`"nbobjects":` *number-of-objects*	
	}	

Figure 8: JSON syntax of the prologue of space id_{space}

then each object content of some given *oid* is preceded by a comment like `//+ob`***oid***, for example an object of oid `_3fzIPzNlWFV01GGQSt` starts with a comment `//+ob_3fzIPzNlWFV01GGQSt` line. The following object content is described in figure 11 below.

@@TODO: **review** and improve ! We use a JSON format to persist our state. Our immutable values could easily be represented in textual syntax, for example a set of three objects of objids `_0iaOiLq4pj20097DNb`, `_1R4TeqlLvhS03o0mGN`, `_7m9EMmdyQKU00euKwB` might be represented as the following JSON object:

```
{  "vtyp"  :  set
   "elem"  :  [    "_0iaOiLq4pj20097DNb",
                   "_1R4TeqlLvhS03o0mGN",
                   "_7m9EMmdyQKU00euKwB" ] }
```

4.2 EBNF Grammar of Data Format

The figure 9 gives the JSON syntax of scalar values or object references in persisted state files. The figure 10 is @INCOMPLETE@ and gives the JSON syntax of composite values in persisted state files. The figure 11 gives the JSON syntax of object contents inside space files.

The syntax of object contents is given in figure 11. The *payload-kind* there is either some C identifier (if it starts with a letter: `A`...`Z` or `a`...`z`) or some objid (if it

$$
\begin{array}{llll}
\textbf{\textit{value}} & \leftarrow & \textbf{\textit{int}} & \text{tagged integers} \\
 & | & \textbf{\textit{float}} & \text{double precision floating point numbers} \\
 & | & \textbf{\textit{string}} & \text{string of Unicode characters enclosed in double quotes} \\
 & | & \textbf{\textit{object}} & \text{reference to mutable } \textit{object} \text{ with a globally unique objid} \\
 & | & \textbf{\textit{set}} & \text{set of ordered unique } \textbf{\textit{object}}\text{s} \\
 & | & \textbf{\textit{tuple}} & \text{set of ordered (and possibly duplicate) } \textbf{\textit{object}}\text{s} \\
 & | & \textbf{\textit{closure}} & \text{function closing over an environment of } \textbf{\textit{value}}\text{s}
\end{array}
$$

$$
\begin{array}{lll}
\textbf{\textit{int}} & \leftarrow & \alpha, \qquad \text{a 63-bit integer represented as an JSON number type} \\
 & & \alpha \in \mathbb{Z}, \\
 & & -2^{62} \leq \alpha \leq 2^{62} - 1
\end{array}
$$

$$
\textbf{\textit{float}} \leftarrow \textit{floating-point-number}, \quad \text{an IEEE 754 double, with a dot}
$$

$$
\begin{array}{lll}
\textbf{\textit{string}} & \leftarrow & \texttt{"}\alpha\texttt{"}, \qquad \text{a UTF-8 string represented as an JSON string type} \\
 & | & \textbf{\{}\ \texttt{"string"}\textbf{:}\ \sigma\ \textbf{\}} \quad \text{when string } \sigma \text{ looks like an objid}
\end{array}
$$

$$
\textbf{\textit{object}} \leftarrow \_\alpha, \quad \text{a Base-62 number prefixed with an underscore}
$$

Figure 9: JSON syntax of scalar values and object references

starts with an underscore _ ...). When it is some C identifier *ident*, an `extern "C"` function (of signature `rpsldpysig_t`, defined in file `refpersys.hh`) named `rpsldpy_`*`ident`* is found by `dlsym(3)` tehn invoked at load time. When it is some objid *objid*, the `rpsldpy`*`objid`* function is called. For example, class objects have `"payload": "class"` as a JSON field in their state file, so are loaded by calling `rpsldpy_class`. If (later) we would have `"payload": "_2j66FFjmS7n03HNNBn"`, then `rpsldpy_2j66FFjmS7n03HNNBn` should be called.

The JSON representation of payloads vary. The figure 12 explains class related payload.

The figure 13 gives the format of mutable set of objects payload. The `"setob":` array might be empty.

The figure **??** gives the format of mutable vector of objects payload. The `"vectob":` array might be empty. Some components could be null.

5 The GUI of REFPERSYS

The GUI of REFPERSYSis built using the Qt5 toolkit. An important consideration in the selection of this toolkit was its C++ API, in addition to it being a stable and mature toolkit.

@@TODO: add more details of windows

set $\leftarrow$ {
```
"vtype": "set",
"elem": [ ω1, ω2, ω3, ...]
}
```
where the ω_i are **object**s represented by *objid*-s

tuple $\leftarrow$ {
```
"vtype": "tuple",
"comp": [ ω1, ω2, ω3, ...]
```
where the ω_i are **object**s represented by *objid*-s

closure $\leftarrow$ {
```
"vtype": "closure"
"fn": ωfun
"env": [ v1, v2, ...]
```
the object giving a function

where the v_i are JSON for closed values.

Figure 10: JSON syntax of composite values

5.1 Menu Items

- The **App** Menu
  - **Dump**: dump the persistent heap
  - **Garbage Collect**: invokes the gargage collector
  - **New Window**: create a new window
  - **Close**: close the current window without dumping
  - **Quit**: quit the application without dumping
  - **Exit**: exit the application after dumping
- The **Help** Menu
  - **About**: display program information

object-content	←	`{`	
		`"oid":` *oid*`,`	the string oid of the current object
		`"mtime":` *modtimeoid*`,`	its modification time
		`"class":` *class-oidoid*`,`	the oid of its class
	[	`"payload":` *payload-kind*]	optional payload kind
	[	`"comps": [` *value* … `],`]	optional components
	[	`"attrs": [` *attr-entry* … `],`]	optional attributes
		@@@ incomplete @@@	
		`}`	
attr-entry	← `{`		
		`"at":` *object*	the oid of the attribute key
		`"va":` *value*	the corresponding attribute value
	`}`		

Figure 11: JSON syntax of object contents inside space files

class-payload	:		
	`"payload": "class",`		
	`"class_super":` *object*`,`	the oid of the superclass	
	`"class_methodict": [`	***method-entry*** … `]`	method dictionnary
method-entry	← `{`		
	`"methosel":` *oid*	the oid of the method selector	
	`"methclos":` *closure*	the corresponding method closure	
	`}`		

Figure 12: JSON syntax of class payload

set-objects-payload	:		
	`"payload": "setob",`		
	`"setob": [`	*oid* … `]`	sorted oids of elements

Figure 13: JSON syntax of mutable set of objects payload

vector-objects-payload	:		
	`"payload": "vectob",`		
	`"vectob": [`	*oid* … `]`	sorted oids of elements

Figure 14: JSON syntax of mutable vector of objects payload

Glossary

Artificial General Intelligence Artificial general intelligence (**AGI**) refers to the specific capacity of a machine to learn and understand any intellectual task that can be performed by a human being. It is the primary goal of some artificial intelligence research, and is sometimes referred to as "strong AI" or "full AI" . 1

metaknowledge Metaknowedge is knowledge about knowledge, and is an inclusive term spanning several disciplines. Bibliography, the academic study of books, and epistemology, the philosophical study of knowledge, are examples of metaknowledge. Even the tagging of documents could be considered as metaknowledge. In AGI, metaknowledge refers to the knowledge about knowledge-based systems. A declarative system might be guided by metarules, that is "expert system" rules to compile or interpret other rules (maybe themselves). . 2

Reflection Reflection is (according to *Wikipedia*) "the ability of a process to examine, introspect, and modify its own structure and behavior", and related to Self-reflection, the capacity of humans (and hopefully of artificial cognitive systems, like REFPERSYS should become) "to exercise introspection and to attempt to learn more about their fundamental nature and essence". . 2

References

[1] Martn Abadi and Luca Cardelli. "An imperative object calculus". In: *Colloquium on Trees in Algebra and Programming*. Springer. 1995, pp. 469–485. URL: `http://lucacardelli.name/Papers/PrimObjImpSIPL.A4.pdf`.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs*. Justin Kelly, 1996. URL: `https://mitpress.mit.edu/sites/default/files/sicp/index.html`.

[3] Andrew W Appel. "Garbage collection". In: *Topics in Advanced Language Implementation* (1991), pp. 89–100. URL: `http://flint.cs.yale.edu/cs421/papers/Appel-GC.ps`.

[4] Henry G Baker. "CONS should not CONS its arguments, part II: Cheney on the MTA". In: *ACM Sigplan Notices* 30.9 (1995), pp. 17–20. URL: http://home.pipeline.com/~hbaker1/CheneyMTA.pdf.

[5] Blaise Barney. POSIX *Threads Programming*. Tech. rep. Lawrence Livermore National Laboratory, 2010. URL: https://computing.llnl.gov/tutorials/pthreads/.

[6] Barry W Boehm. "A spiral model of software development and enhancement". In: *Computer* (1988), pp. 61–72.

[7] Jean-Pierre Briot and Pierre Cointe. "A Uniform Model for Object-Oriented Languages Using the Class Abstraction." In: *IJCAI*. Vol. 87. 1987, pp. 40–43. URL: https://pdfs.semanticscholar.org/737c/69fc33a7ff6232a59556bdc9cc98 pdf.

[8] Frederick P. Brooks Jr. "No Silver Bullet - Essence and Accidents of Software Engineering". In: *Computer* 20.4 (Apr. 1987), pp. 10–19. ISSN: 0018-9162. DOI: 10.1109/MC.1987.1663532. URL: https://doi.org/10.1109/MC.1987.1663532.

[9] Frederick P. Brooks Jr. *The Mythical Man-month (Anniversary Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-83595-9.

[10] Craig Chambers, David Ungar, and Elgin Lee. "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes". In: *Lisp and symbolic computation* 4.3 (1991), pp. 243–281. URL: http://bibliography.selflanguage.org/_static/implementation.pdf.

[11] Eugene Charniak. *Introduction to deep learning*. The MIT Press, 2019. ISBN: 978-0-262-03951-2.

[12] Frédéric Chazal. *High-dimensional topological data analysis*. 2016. URL: https://hal.inria.fr/hal-01316989/document.

[13] Pierre Cointe. "Metaclasses are first class: the OBJVLISP model". In: *ACM Sigplan Notices*. Vol. 22. 12. ACM. 1987, pp. 156–162. URL: https://www.researchgate.net/profile/Pierre_Cointe/publication/221320901_Metaclasses_are_First_Class_the_ObjVlisp_Model/links/5422e2d50cf290c9e3ae2016/Metaclasses-are-First-Class-the-ObjVlisp-Model.pdf.

[14] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.

[15] Jean-Luc Dormoy and Sylvie Kornman. "Meta-knowledge, autonomy, and (artificial) evolution: some lessons learnt so far". In: *Toward a Practice of Autonomous Systems: Proceedings of the First European Conference on Artificial Life*. MIT Press. 1992, p. 392. URL: http://dormoy.org/JLuc/Papers/ECAL91.pdf.

[16] Ulrich Drepper. *How to write shared libraries*. Tech. rep. 2011. URL: https://www.akkadia.org/drepper/dsohowto.pdf.

[17] Jean-Marc Fouet and Basile Starynkevitch. *Describing control*. Tech. rep. FR8801379/CEA-CONF–9239. CEA, Aug. 1987. URL: https://inis.iaea.org/collection/NCLCollectionStore/%5C_Public/19/059/19059867.pdf.

[18] Russell Greiner and Douglas B Lenat. "A Representation Language Language". In: *AAAI*. Vol. 1. 1980, pp. 165–169. URL: http://new.aaai.org/Papers/AAAI/1980/AAAI80-047.pdf.

[19] Carolina Hernández Phillips et al. "Challenges in Debugging Bootstraps of Reflective Kernels". In: *IWST19 - International workshop on Smalltalk Technologies*. Cologne, Germany, Aug. 2019. URL: https://hal.archives-ouvertes.fr/hal-02297710.

[20] Douglas R Hofstadter. *I am a strange loop*. Basic books, 2007. ISBN: 978-0-465-03078-1.

[21] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY, USA: Basic Books, Inc., 1979. ISBN: 0465026850.

[22] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC, 2016. URL: http://gchandbook.org/.

[23] Alan C Kay. "The early history of Smalltalk". In: *History of programming languages—II*. ACM. 1996, pp. 511–598. URL: http://esug.org/data/HistoricalDocuments/Smalltalk80/SmalltalkHistory.pdf.

[24] Richard Kelsey, William Clinger, Jonathan Rees, et al. "Revised 5^{th} report on the algorithmic language Scheme". In: (1998). URL: https://schemers.org/Documents/Standards/R5RS/.

[25] Sanjay Kumar and Rajkishore Prasad. "Importance of expert system shell in development of expert system". In: *International journal of innovative research & development* 4.3 (2015), pp. 128–133. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.854.6373&rep=rep1&type=pdf.

[26] Douglas B Lenat. "Theory formation by heuristic search: The nature of heuristics II: background and examples". In: *Artificial Intelligence* 21.1-2 (1983), pp. 31–59. URL: https://blog.funcall.org/docs/am-eurisko/Heuristics_II.pdf.

[27] Douglas B. Lenat. "Eurisko: A Program That Learns New Heuristics and Domain Concepts". In: *Artif. Intell.* 21.1-2 (Mar. 1983), pp. 61–98. ISSN: 0004-3702. DOI: 10.1016/S0004-3702(83)80005-8. URL: http://dx.doi.org/10.1016/S0004-3702(83)80005-8.

[28] Douglas B. Lenat and R. V. Guha. "The Evolution of CycL, the Cyc Representation Language". In: *SIGART Bull.* 2.3 (June 1991), pp. 84–87. ISSN: 0163-5719. DOI: 10.1145/122296.122308. URL: http://doi.acm.org/10.1145/122296.122308.

[29] Josh Lerner and Jean Tirole. *The Simple Economics of Open Source*. Working Paper 7600. National Bureau of Economic Research, Mar. 2000. DOI: 10.3386/w7600. URL: http://www.nber.org/papers/w7600.

[30] John R. Levine. *Linkers and loaders*. Morgan-Kauffman, Oct. 1999. ISBN: 1-55860-496-0. URL: https://www.iecc.com/linker/.

[31] Jacob Matthews and Robert Bruce Findler. "An operational semantics for R5RS Scheme". In: *Proceedings of the Sixth Workshop on Scheme and Functional Programming*. Citeseer. 2005, pp. 41–54. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.118.9251&rep=rep1&type=pdf#page=47.

[32] Nancy Mazur. "Compile-time garbage collection for the declarative language MERCURY". In: (2004). URL: https://lirias.kuleuven.be/retrieve/77222.

[33] Jean-Marc Nigro and Yann Barloy. "The meta-inferences engine: A new tool to manipulate metaknowledge". In: *Knowledge-Based Systems* 21.7 (2008), pp. 588–598. URL: http://www.math.s.chiba-u.ac.jp/~yasuda/open2all/Paris06/IPMU2006/HTML/FINALPAPERS/P437.PDF.

[34] Cathy O'Neil. *Weapons of math destruction: How big data increases inequality and threatens democracy*. Broadway Books, 2016. ISBN: 9780553418811. URL: https://en.wikipedia.org/wiki/Weapons%5C_of%5C_Math%5C_Destruction.

[35] Jacques Pitrat. *A Step toward an Artificial Artificial Intelligence Scientist*. Tech. rep. CNRS and LIP6 Université Paris, 2009. URL: https://pdfs.semanticscholar.org/2117/9600b3f05c0af399f9acbfc6e7b6d24daf03.pdf.

[36] Jacques Pitrat. *Artificial Beings: The Conscience of a Conscious Machine*. Wiley ISTE, 2009. ISBN: 978-1-848-21101-8.

[37] Jacques Pitrat. "Implementation of a reflective system". In: *Future Generation Computer Systems* 12.2 (1996), pp. 235–242. ISSN: 0167-739X. DOI: https://doi.org/10.1016/0167-739X(96)00011-8. URL: http://www.sciencedirect.com/science/article/pii/0167739X96000118.

[38] Jacques Pitrat. *My view on Artificial Intelligence*. blog. 2013-2019. URL: http://bootstrappingartificialintelligence.fr/WordPress3/.

[39] Christian Queinnec. "Continuations and Web Servers". In: *Higher Order Symbol. Comput.* 17.4 (Dec. 2004), pp. 277–295. ISSN: 1388-3690. DOI: 10.1007/s10990-004-4866-z. URL: https://core.ac.uk/download/pdf/81910787.pdf.

[40] Christian Queinnec. *Lisp in Small Pieces*. New York, NY, USA: Cambridge University Press, 1996. ISBN: 0-521-56247-3. URL: https://pages.lip6.fr/Christian.Queinnec/WWW/LiSP.html.

[41] Christian Queinnec. *Lisp in small pieces*. Cambridge University Press, 2003.

[42] Jon Rafkind et al. "Precise Garbage Collection for C". In: *Proceedings of the 2009 International Symposium on Memory Management*. ISMM '09. Dublin, Ireland: ACM, 2009, pp. 39–48. ISBN: 978-1-60558-347-1. DOI: 10.1145/1542431.1542438. URL: http://doi.acm.org/10.1145/1542431.1542438.

[43] John C. Reynolds. "The Discoveries of Continuations". In: *Lisp & Symbolic Computation* 6.3-4 (Nov. 1993), pp. 233–248. ISSN: 0892-4635. DOI: 10.1007/BF01019459. URL: http://dx.doi.org/10.1007/BF01019459.

[44] Manuel Serrano and Pierre Weis. "Bigloo: a portable and optimizing compiler for strict functional languages". In: *International Static Analysis Symposium*. Springer. 1995, pp. 366–381. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.8424&rep=rep1&type=pdf.

[45] Basile Starynkevitch. "A Middle End Lisp Translator for GCC". In: *GCC Research Opportunities workshop*. 2009. URL: http://www.doc.ic.ac.uk/~phjk/GROW09/papers/01-MELT-Starynkevitch.pdf.

[46] Basile Starynkevitch. "Expliciter et utiliser les données et le contrôle pour les connaissances par des metaconnaissances : systèmes EUM1 et EUM2". [explicitation and use of data and control for knowledge with metaknowledge : the EUM1 and EUM2 systems]. PhD thesis. Université Paris 6 - LIP6, Dec. 1990. URL: http://www.theses.fr/1990PA066799.

[47] Basile Starynkevitch. *GCC MELT website (archive)*. (archive of the old gcc-melt.org site). 2008-2016. URL: http://starynkevitch.net/Basile/gcc-melt/.

[48] Basile Starynkevitch. "MELT - a Translated Domain Specific Language Embedded in the GCC Compiler". In: *DSL2011 IFIP conf.* Bordeaux (France), Sept. 2011. arXiv: 1109.0779 [cs.PL]. URL: http://adsabs.harvard.edu/abs/2011arXiv1109.0779S.

[49] Basile Starynkevitch. "Multi-Stage Construction of a Global Static Analyzer". In: *GCC summit*. GCC. Ottawa, Canada, July 2007, pp. 143–152. URL: http://starynkevitch.net/basile/gccsummit2007-starynkevitch.pdf.

[50] Basile Starynkevitch. *Specialized Static Analysis tools for more secure and safer IoT software development. Bismon documentation*. Tech. rep. work in progress, H2020 draft deliverable. CEA/LIST, Oct. 2019. URL: http://starynkevitch.net/Basile/bismon-chariot-doc.pdf.

[51] Christian F. Tschudin and Lidia Yamamoto. "Harnessing Self-modifying Code for Resilient Software". In: *WRAC*. 2005.

[52] Steven Weber. *The Success of Open Source*. Cambridge, MA, USA: Harvard University Press, 2004. ISBN: 0674012925.

[53] Paul R Wilson. "Uniprocessor garbage collection techniques". In: *International Workshop on Memory Management*. Springer. 1992, pp. 1–42. URL: http://3e8.org/pub/scheme/doc/gcsurvey.pdf.

[54] Shoshana Zuboff. "Big other: surveillance capitalism and the prospects of an information civilization". In: *Journal of Information Technology* 30.1 (2015), pp. 75–89. URL: https://cryptome.org/2015/07/big-other.pdf.

Index