Why a New Operating System?

François-René Rideau Đặng-Vũ Bân
<fare@tunes.org> *

2nd February 2003

Abstract

In this paper, we propose a reconstruction of the general architecture of an operating system. In a first part, we start from the very principles of cybernetics, and study the general nature, goals and means of operating systems. In a second part, we examine the crucial problem of the expressiveness of a computing system, conspicuously comprising both operating system and programming language. In a third and final part, we focus on particular services commonly found in operating systems, and criticize current designs at the light of the previous theory. All along, we sadly find that existing operating systems are deeply flawed, due to both historical and political reasons; happily, the advent of Free Software removes the obstructions to progress in this matter.

*This article was initially started as a manifesto for the *Tunes* project down in early 1995. For years, only the main arguments in the first part were laid down and explained. I still haven't finished it.

Contents

1 Introduction

This paper aims at consistently demonstrating, out of sufficiently clear definitions, that while currently available computing system software provide a lot of *expedient* services, their low-level structure forbids them to provide *useful* services, which leads to huge, inefficient, slow, unusable, unportable, unmaintainable, unupgradeable, software. This paper tries to explain why the current design of "system software" is deeply and unrecoverably flawed, and proposes a new way for designing computing systems such as to achieve real utility. The proposed design method does not require but well-known, available, though sometimes unjustly deprecated, technologies.

2 Operating Systems and Utility

2.1 Utility

Between a good and a bad economist this constitutes the whole difference - the one takes account of the visible effect; the other takes account both of the effects which are seen, and also of those which it is necessary to foresee. Now this difference is enormous, for it almost always happens that when the immediate consequence is favourable, the ultimate consequences are fatal, and the converse. Hence it follows that the bad economist pursues a small present good, which will be followed by a great evil to come, while the true economist pursues a great good to come, - at the risk of a small present evil.

– Frederic Bastiat, *That Which is Seen, and That Which is Not Seen* [2]

We herein call **useful** something that saves time, effort, money, strength, or anything valuable *in the long run* and *for a lot of people*. Utility is strictly opposed to Harmfulness, but we also oppose it to mere Expediency something being called expedient if it saves such valuable things, but most usually only in the short term, for special, personal, temporary purposes, and not (forcibly) for general, universal or permanent purposes.

Utility and Expediency are relative, not absolute concepts: how much you save depends on a reference, so you always compare the utility of two actions, even though one of the actions might be implicit. Utility of an isolated, unmodifiable, action is therefore meaningless. Particularly, *from the point of view of present action*, utility of past actions is a meaningless concept; however, the study of the utility that such actions may have had when they were taken can be quite meaningful. More generally, *Utility is meaningful only for* **projects***, never for* **objects**. Projects here must not be understood in the restricted meaning of conscious projects, but in the more general one of a consistent, lasting, dynamic behavior.

Note that projects can be considered in turn as objects of a more abstract "meta-" system; but the utility of the objectized project becomes itself an object of study to (an extension of) the metasystem, and should not be confused with the utility of the studying metasystem. Sciences of man and nature (history, biology, etc) lead to the careful study of terrifying events and dangerous phenomena, but the utility of such study is proportional rather to some kind of relevance or amplitude of the studied projects, than to their utility from the point of view of their various actors.

Utility is a moral concept, that is, a concept that allows pertinent discourse on its subject. More precisely, it is an ethical concept, that is, a concept colored with the ideas of Good and Duty. It directly depends on the goal you defined for general interest. Actually, Utility is as well defined by the moral concept of Good, as Good is defined by Utility; to maximize Good *is* to maximize Utility.

Like Good, Utility needs not be a totally ordered concept, where you could always compare two actions and say that one is "globally" better than the other. Utility can be made of many distinct, sometimes conflicting criteria. Partial concepts of Utility can be refined in many ways to obtain compatible concepts that would solve more conflicts, gaining separation power, but losing precision.

However, a general theory of Utility is beyond the scope of this article (those interested can find a first sketch in J.S.Mill's "Utilitarianism" [6], and a more refined discussion in Henry Hazlitt's "The Foundations of Morality" [5]). Therefore, we'll herein admit that all the objects previously described as valuable (time, effort, etc) are indeed valuable as far as general interest is concerned.

2.2 Information

Obviously, a man's judgement cannot be better than the information on which he has based it. Give him the truth and he may still go wrong when he has the chance to be right, but give him no news or present him only with distorted and incomplete data, with ignorant, sloppy or biased reporting, with propaganda and deliberate falsehoods, and you destroy his whole reasoning processes, and make him something less than a man.

– Arthur Hays Sulzberger

Judgements of Utility deeply depend on the knowledge of the project being judged, of its starting point, of its approach. Now, humans are no gods who have universal knowledge to base their opinions upon; they are no angels who by super-natural ways, receive infuse moral knowledge. Surely, many people believed it, and some still do. But everyone's personal experience, and mankind's collective experience, History, show how highly improbable such things are. Without any further discussion, we will admit an even stronger result: that, by the finiteness of the structure of the human brain, any human being, at any moment, can only handle a finite amount of *information.*

This concept of information should be clarified. The judicial term from the Middle Ages slowly took the informal meaning of the abstract idea of elements of knowledge; it was only with seventeenth century mathematicians that a formal meaning could timidly appear, that surprisingly found its greatest confirmation in the nineteenth century with thermodynamics, a branch of physics that particularly studied large *dynamical systems.* Information could thus be formalized as the opposite of the measurable concept of entropy. The information we have irreversibly decreases as we look forward or backward in time, beyond the limits of our knowledge, on this side of these limits being present. That is, information is a timely notion, valid only in dynamical systems. And such is Life, a dynamical system.

As for Utility before, there needs not be a universal total ordering on Information; what we most often have is partial orderings, and each of us has to try arbitrarily choose finer orderings when basing a decision upon equivocal information. For information is also an moral concept, though it is not until late twentieth century, with cybernetics, that the deep relationship between information and morals *explicitly* appeared. Few people remember cybernetics as something else than a crazy word associated to the appearance of information technology, but we invite the reader to consult the original works of Norbert Wiener on the subject [7]. However, this relationship had been implicitly discovered by liberal economists of the eighteenth and nineteenth centuries, then rediscovered by biologists studying evolution, and surely, many have always intuititively felt this relationship. What allowed to make it explicit might be the relativization of ethics as something that was not to be taken as known and granted, but first as unknown and more recently as incomplete.

Moral judgments depend on the information we have, so that in order to make a better judgement, we must gather more information. Of course, even though we might have rough ways to quantify information, this doesn't make elements of information of same quantity interchangeable; What information is interesting depends on the information we already have, and on the information we can expect to have. Now, gathering information itself has a cost, that physicists may associate to free energy, which is never zero, and must be taken into account when gathering information.

Because the total information that we can handle is limited, any inadequate actual information that be gathered would be to the prejudice of more adequate potential information. Such inadequate information is then called *noise*; noise is worse than lack of information, because it costs resources that won't be available for adequate information. Thus, in our dynamic world, the quest of information itself is subject to criteria of utility, and the utility of information is its *pertinency*, its propensity to favorably influence moral judgements. As an example, the exact quantization of information, when it is possible, itself requires so much information, that it is seldom worth to be sought. Of course, pertinency in particular is not more an absolute concept than utility in general. When a criteria for pertinency is implicitly available, we might use the term "Information" for raw information, and "Knowledge" for pertinent information.

So to gather information in better ways, one must scan the widest possible space of elements of knowledge, which is the essence of *Liberty*; but the width of this knowledge must be measured not in terms of its cost or of its interest in case it was true, but in terms of its pertinency and of its solidity, which is the essence of *Security.*

These are dual, inseparable aspects of Knowledge, that get their meaning from each other. Any attempt to priviledge one upon the other is pointless, and in the past and present, such attempts have led to many a disaster: trying to promote some liberty without corresponding security leads to chaos, whereas promoting security without associated liberty leads to totalitarianism.

Reality and potentiality, finiteness of knowledge, world as a dynamic system, relation between information and decision, duality of liberty and security, all these are parts of a consistent (we hope) approach of the world, that we will admit in the rest of this paper, at least on the considered subjects. A more detailed study of these moral issues per se would certainly be quite interesting, but the authors feel that such study ought to be postponed to another paper, and invite readers to refer to the bibliography (and contribute to it), so as to focus on the goal of this article, discussion about Computer Systems, whereas these moral concepts are a means.

2.3 Computers

The highest goal of computer science is to automate that which can be automated.

– Derek L. VerLee

Computers are machines that handle quickly large amounts of exact discrete information, and interact with the external world, according to a set of exact discrete directives called "programs". This makes them most suited to apply concepts from the above-mentioned information theory; everywhere else in life, information is approximate continuous, and difficult to quantize. Actually, the histories of information theory and of computers, that are information technology, are deeply interrelated; but these histories escape the subject of this article. Just note that being a computer is an abstract concept independent from the actual implementation of a computer: if most current computers are made of silicon transistors, their ancestors were made of metallic gears, and no-one knows what their remote successors will be made of.

Computers are built and programmed by men and for men, with earthly materials and purposes. Hence the utility of computers, among other characteristics, is thus to be measured like the utility of any object and project in the human-reachable world, relatively to men. And, because what computers deal with is information, their utility lies in what will allow humans to access more information in quicker and safer ways, that is to communicate better through them computers with other humans, with nature, with the universe.

Again, Utility criteria should not only compare the visible value of objects, but also their *cost*, often invisible, in terms of what objects where discarded for it. Cost of computer information includes the time and effort spent at manufacturing or earning money to buy computer hardware and software, but also the time and effort spent before the computer to explain it the work you want to be done, and the time and effort spent verifying, correcting, trying again the obtained computer programs, or just worrying about the programmed computer crashing, preparing for possible or eventual crashes, and recovering from these. All this valuable free energy might have been spent much more profitably at doing other things, and is part of the actual cost of computerware, even when not taken into account by explicit financial contracts. We will stress on this point later.

So to see if computers in general are a useful tool, we can take the lack of computer as the implicit reference for computer utility, and see how computers benefit or not to mankind, comparing the result and cost. *Once properly programmed*, computers can do quickly and cheaply large amounts of simple calculations that would have required a large number of expensive human beings to manage (which is called "number crunching"); and they can repeat relentlessly their calculations without committing any of those mistakes that humans would undoubtly have made. When connected to "robot" devices, those calculations can replace the automatic parts of work, notably in the industry, and relieve humans from the degrading tasks of chain work, but also control machines that work in environments where no human would survive, and do all that much more regularly and reliably than humans would do. Only computers made possible the current state of industry and technology, with automated high-precision mass production, science of the very small, the very large, and the very complex, that no human senses or intelligence could ever have approached otherwise.

Thus, computers save a great amount of human work, and allow things that no amount of human work could ever bring without them; not only are they useful, but they are necessary to the current state of human civilization. We let the reader meditate on the impact of technology on her everyday life, and compare it to what was her grandmother's life. That this technology may be sometimes misused, and that the savings and benefits of this technology be possibly misdistributed, is a completely independent topic, which may hold for quite any technology, and which will not be otherwise commented in this article.

2.4 Limits of Computers

Some only see in computer's utility a matter of raw performance, a quantitative progress, but not a qualitative one, at least, nothing qualitatively better than what other tools bring about. However we already saw that beyond their performance, beyond the volume of information handled and the speed at which it is handled, which already suffice to make computers a highly desirable tool, computers bring something fundamentally more important than raw information or raw energy, something that is seldom explicitly acknowledged: a new kind of reliability that no human effort can achieve.

Not only can computers perform tasks that would require enormous amounts of human work without them, and do things with more precision than humans, but they do it with reliability that no human can provide. This may not appear as very important, not even as obvious, when the tasks undertaken are independent one from the other, when erroneous results can be discarded or will be compensated somehow by the mass of good results, or when on the contrary the task is unique and completely controlled by one man. But when the failure of just one operation involves the failure of the whole effort, when a single man cannot warranty the validity of the task, then computers prove inestimable tools by their reliability.

Of course, computers are always subject to failures of some kinds, to catastrophes and accidents; but computers are not worse than anything else with respect to such events, and can be arbitrarily enhanced in this respect, because their technology is completely controlled. However, not only is it *not* a problem particular to computers, but computers are most suited to fight this problem: unpredictable failures are the doom of the world as we live, where we always know a tiny finite piece of information, so even if we can sometimes be *fairly sure* of many things, and can never be *completely sure* about anything, as we can never *totally* discard the event of some unexpected external force significantly perturbating our experiments. The phenomenon is the most pronounced with humans, where every individual is such a complex system by himself, that one can never control all the parameters that affect him, can never perfectly reproduce them; so there are always difficulties in trusting a man's work, even when his sincerity is not in doubt. On the contrary, by their very mechanical nature of their implementation, by the exactitude of their computations, which derives from their very abstract design principle, computing is both a predictable and a reproducible experiment; it can be both mathematically formalized, and studied with the tools of the physicists and engineers; computer behavior is both producible and reproducible at will; and this founds computer reliability: you can always check and counter check a computer's calculations, experiment with them under any condition one requires before one will trust them.

We see that computers allow to accumulate reliability like nothing else in the human-reachable world, though this reliability must be earned in the hard way, by ever-repeated proofs, checks, and tests. In fact, this reliability is one of the two faces of information, which is what information technology is all about, of which computers as we know them are the current cutting edge.

The problem with computers, the absolute limit to their utility, is that by the same mechanical virtues that make us trust their answers as the result of understandable and double-checkable formal computations, they somehow can't *create* information that isn't formally derivable from their input, and in as much as we introduce randomness and heuristics in their input to simulate more creative behavior, we are not able to trust the answers anymore. That is, computers may reveal reveal potential trust in existing information; they may build trustable information from previous trustable information; they may generate untrustable information at random; but they cannot generate new trustable information about the external world, least it be tautological. Any new trustable information that lies in a computer must derive through natural laws of logic from the work of men who built and programmed the computers, and from the external world with which the computer interacts by means of sensors and various devices.

Hence the limits of computers are men: what they can program in a computer, what the devices they hook into computers produce, what they teach the computer to do with all the input. If a man misdesigns or misprograms a computer, if he feeds it with improper data, if he puts it in an environment not suitable for correct computer behavior, then the computer cannot be expected to yield any correct result. One can fully trust everything he sees and tests about a computer, but as computers grow in utility and complexity, there are more and more things one cannot personally see and test about them, so one must rely on one's fellow human beings to have checked them. Again, this is not worse than anything else in the human world; but for computers as well as for anything else, these are hard limits of which we should all be conscious.

2.5 Computing as a Project

Man is surely a limit to the power of computers, in that computers are made by man, and can be no better than man makes them. But this is not to be understood individually, as computers are not each the work of one man, but are collectively the work of mankind. Computing is a global project.

Like any product of civilization, computers depend on a very rich technological, economical, and social context, before they can even exist, not to talk about their being useful. They are not the work of a single man, who would be born naked in a desert world, and would build every bit of them from scratch. Instead, they are the fruit of the slow accumulation of human work, of which the foundations of civilization participate at least as much as the discoveries of modern scientists. The context is so necessary, that most often it is implicit; but one shouldn't be mistaken by this implicitness and forget or deny the necessity of the context. Actually, this very context, result of this accumulation process, is what Civilization is.

But again, the characteristic of information technology, is that the information you manage to put in it can be made to decay extremely slowly, as compared to objects of same energy: we can expect data entered today in a computer, that is interesting enough to be copied once every ten years at least, to last as long as information technology will exist, that is, as long as human civilization persists. Of course, huge monuments like the egyptian pyramids are work of men that decay slowlier, need less care, and resist to harsher environments, so may last longer; but their informative yield is very weak, as compared to their enormous cost. If only slowness to decay was meant and not informational yield, then nuclear wastes would be the greatest human achievement!

Now computing has the particularity, among the many human collective projects, and as part of mankind being its own collective project, that it can be contributed to in a cumulated way for years. For this reason, we can have the greatest hope in it, as a work of the human race, as a tool to make masterpieces last longer, or as a masterpiece of its own. Computing has already gained its position among the great inventions of Man, together with electricity, writing, and currency.

This whole paper tackles the problems of software as an evolving domain. If ever software settles and stabilizes, or comes to a very slow evolution, then the phenomena described in this paper may cease to be dominant in the considered domain. However, because life is movement, as long as there will be life, there will be a domain where these phenomena are of importance. Besides, the authors are confident that computer software, whatever it will be like, will be a lively domain until it possibly reaches AI.

2.6 Computing versus Artificial Intelligence

Alan Turing thought about criteria to settle the question of whether machines can think, a question of which we now know that it is about as relevant as the question of whether submarines can swim.

– E.W. Dijkstra

Let us justify the persistence of Computing as a Project, even when faced with this alleged doom of it: Artificial Intelligence.

Many dream, hope, worry, predict or otherwise expect that some day, the cumulated work invested in computing will allow humans to create some computer-based being, which they call “artificial intelligence”, or more simply, AI. Such AIs would rival with their human creators as for “intelligence”, that is, their creativity, their ability to undertake independently and voluntarily useful projects; they dream (or some of them have the nightmare) that mankind engenders a next step in evolution by non-genetical means. According to some people, such AI would be the End of Computing as a Project, since humans wouldn’t need to program anymore, leaving the task to AIs.

Now, should this dream come true (the eventuality of which won’t be discussed in this article), by Information Theory’s version of Cantor’s diagonal argument, the workings of AIs must globally surpass the understanding of AIs themselves, and hence of humans, if the AIs are similarly endowed as humans. This holds even though the general principles behind the functioniong of AIs might be understood: as with physics, the knowledge of elementary laws doesn’t imply understanding of whole phenomena, for the formidable context involved in anything but the simplest applications (and the most useless, as far as “intelligence” is meant) would make it impossible for the most developed human or artificial brain to apprehend.

The latter argument does not question the possibility of AI as an eventual human work: there is plenty of evidence that systems governed by a few human-understandable, human-enforceable rules can generate ununderstandable chaotic behavior[1]. Rather, the argument means that if we replace in all the current discussion “human” by “sentient”, with AIs being a new kind of different (superior or not) sentient beings, the situation of computing would remain essentially the same.

Indeed, Computing is an activity characterized by exact formalizability and as complete understanding as desired of running programs, with the choice and evolution of the programs being directed by human (sentient) beings.

AI, if it ever appears, will not quite be computing as we know it anymore, yet will need Computing even more than we do now.

Maybe this AI will use a computer as an underlying structure, and will need most advanced computing techniques to be deployed; but the AI itself will not be a computer as we defined it, and querying the AI will not be computing anymore, though some may think that the ultimate goal of computing be to transcend computing in such way.

Anyway, current design of computing systems, as we will show, greatly limits the potential of computer software into what a few programmers can fully understand; hence, until this design is replaced, AI will stay a remote dream. And even when and if this dream comes true, the problems we describe may be food for thought for the AIs that would replace current human readers. Computing is will always be a Project for sentient beings, be them AIs instead of humans.

2.7 Computing Systems

Computer Science is no more about computers than astronomy is about telescopes.

– E. W. Dijkstra

We herein call Computing System any dynamic system where what interests us is the exact information contained by part of it. Note that a comput**ing** system is not quite the same as a comput**er** system. In a comput**er** system, the computer is a static tool used in the project, but not part of it. In a comput**ing** system, the computer (or most probably only its program) is the very dynamic project being considered. Computer systems have been the subject of study of many very

[1]Even simple purely mathematical objects like the digits of number π or the Mandelbrot set have behavior beyond comprehension, not to talk about Chaitin’s “number of wisdom” Ω [3]. Physical systems are even harder to comprehend, and the most precise computer simulations of even the simplest non-linear dynamic systems quickly diverge into chaos. Finally, intrinsicly ununderstandable is the chaos that results from competitive interaction with a large number of sentient, as witnesses the stock market, for instance, for any perceived regularity self-defeatingly gathers against it the behavior of those who perceive it.

All in all, there are many sources of chaos, from mathematical complexity, physical randomness, competition with other systems and interaction with human beings, that could give birth to “intelligent” behavior in machines. The whole problem is for AI researchers to develop tools to identify and harness this chaos, so as to take advantage of it.

proficient people, who have published a great number of most interesting books on it. Computing systems are the subject of this article, upon which we'll try to bring new lights.

As an example, a given modern washing machine is often a very useful comput**er** system, where a static program manages the operations; but its utility lies entirely in the washing of clothes, so that as a comput**ing** system, it is not excessively thrilling. The development of washing machines, on the other hand, contains a computing subsystem of its own, which is the development of better washing programs; this computing system might not be the most exciting one, but it is nevertheless an interesting one.

Similarly, a desktop computer alone might be a perfect computer system, it won't be a very interesting computing system until you consider a human, perhaps one among many, sitting in front of it and using it. And conversely, a man alone without computer might have lots of ideas, he won't constitute a very effective computing system until you give him the ability to express it into computer hardware or software. Note that desktop publishing in a business office *is* considered as being some kind of software, but that, as long as this information is not spread, copied and manipulated much by computers, as long as the writing is very redundant but not automated, it is not a very interesting computing system. Developing tools to automate desktop publishing, on the other hand, is an interesting computing system; even desktop publishing, if it allowed to take any tiny active part in the development of such tools, would be an interesting computing system; unhappily, there is a quasi-monopoly of large corporations on such development, that greatly restricts the amount of computing in that system, which we'll investigate in following chapters.

A most interesting Computing System, which particularly interests us, is made of all interacting men, computers, and particles in the Universe, where the information being considered is all that encoded by all known computers; we may call it the Universal Computing System (UCS). Actually, as the only computers we know in the Universe are on Earth, or not far from it in space, it is the same as what we might call the Global Computing System (GCS); however the two might diverge from each other in some future, so let's keep them separate.

Now, the question that this article tries to answer is "how to maximize the utility of the Universal Computing System ?". That is, we take the current utility of computers for granted, and ask not how they can be useful, but how their utility can be improved, maximized. We already saw that this utility depends on the amount of pertinent information such systems yield as well as the free energy they cost. But to answer this question more precisely requires at the same time a general study of Computing Systems in general, of the way in which they are or should be organized, and a particular study of current, past, and expected future computing systems, that is, where the Universal Computing System is going if we're not doing anything.

2.8 Subsystems

When studying a dynamic system, one must always place oneself in an external, "meta" system, and choose some "representation" of the studied system. What kind of meta-system(s) and representation(s) to choose is a difficult question; again, the representations are better that allow to extract more information from the study of the system, which needs not be a total ordering among representations.

Particularly, one could try to formalize the UCS with the set of the physical equations of its constituting particles. While such thing might be "theoretically possible", the complexity of the obtained equations would such that any direct treatment of them would be impossible, while the exact knowledge of these equations, and of the parameters that appear in it, is altogether unreachable. Thus, this formalization is not very good according to the above criterion.

A fundamental tool in the study of any system, dynamic or not, called analysis, consists into dividing the system into individual subsystems, such that those subsystems, and the interactions between those subsystems be as a whole as simple as possible to formalize. Note that these subsystems need not (and often should not) form an homogeneous mass of (quasi-)isomorphic systems; on the contrary, the richness of information in the total system will depend on the fact that every subsystem be specialized in its way, and doesn't waste its resources by merely being redundant with its neighbours.

For computing systems, the basic, obvious though not sole possible analysis is to consider computers and their human users as the individual subsystems. Because information flows quickly and broadly inside each of these subsystems, but comparatively slowly and tightly between them, they can be considered as decision centers, each of which takes actions depending mostly on its internal information, and slowly interacting with each other "on purpose" (because according to these internal informations).

Humans interact with other humans and computers; computers interact with other computers and humans. But while the stable, exact, objectized information lies in the computers, the dynamic nature of the project can be traced down to the humans; thus, even though only the computerized information might be ultimately valuable to the computing system, the information flow among humans, is a non-negligible aspect of the computing system as a project.

Surely, this is not the only possible way to analyze computing systems; but it is a very informative one, and any "better" analysis should take all of this into account. For instance, one relevant approach is to refine the subdivision of computer activities according not just to corresponding individual human computer users of these activities, but according to division of trust between these humans under the various roles that they assume: the same person may assume several roles during his computer life, and the trust one places in various programs (including those developed by oneself while assuming another role) varies according to these roles.

Anyway, the point is that what counts when analyzing a system is the ability of the analysis to yield relevant information at a competitive cost. A "canonical representation" in terms of atoms and waves, while possibly being a valid analysis of a system, needs not be the most interesting one. Computers may be made, from the hardware, physical point of view, of electronic semiconductor circuitry and other substratum; from the information point of view, this is just transient technological data; tomorrow's computers may be made of a completely different technology, they will still be computers. Similarly, living creatures, among which humans, are, as far as we know, made of organic molecules; but perhaps on other places in the universe, or in other times, things can live that are not made of the same chemical agents (actually, there is genetic variation in the molecular composition of even known living creatures).

What makes creature living is not the matter of which it is made (or else, the soup you obtain after reducing a man to bits would be as living as the man). What makes the living creature is the structure of internal and external interaction that the layout of matter implements. A chair is not a chair because it's made of wood or plastic, but because it has such a shape that a human can sit on it. What makes the thing what you think it is, are abstract patterns that make you recognize it as such, that constitute the idea of the thing. And as for computing systems, the idea lies in the flow of information, not the media on which the information flows or is stored.

2.9 Operating Systems

Often in a discussion, I will ask the other person to define some term. It is not that I believe that terms are absolute, and want to test whether the person knows its One True Meaning. On the contrary, words are conventions, and it is necessary to negociate a common meaning so a sane discussion be possible. For a constructive discussion *is* a negociation.

– Faré

Now, we can define what an *Operating System* is (for which we use the acronym *OS*), that the project of this article is all about.

Given a collection of subsystems of a cybernetical systems, we call "Common Background" the information that we can expect every of these subsystems to have. For instance, if we can expect most Europeans to wear socks, then this expectation is part of the Common Background of Europeans. If we can expect *all* the computers we consider to use binary logic, then this fact is part of the Common Background for those computers. This Common Background can thus contain both established facts and probabilistic expectations. The Common Background for a collection of human beings is called their collective culture, or even their Civilization, if a large, (mostly) independent collection of human beings is considered. The common background for a collection of computers is called their *Operating System.*

The concept of Common Background appears in any cybernetical system where a large enough number of similar subsystems exist. Common Backgrounds grow in complexity only if those subsystems do get more complex too, and the large number of such systems means that these should be self-replicating, or more precisely correlated to self quasi-replication. To sum it up, an interesting concept of Common Background is most likely to appear only when some kind of "life" has developed in the cybernetical system, or when we're examining a large number of specifically considered similar systems.

Note that the "similarity" between the subsystems tightly corresponds to the existence of information common to the subsystems, that constitute the Common Background. In no way does this similarity necessarily correspond to any kind of "equality", among the subsystems: how could two subsystems be exactly the same, when they were specifically considered as disjoint subsystems, made of different atoms ? The similarity is an abstract, moral, concept, which must be relative to the frame of comparison that makes the considered information pertinent; a moral frame of Utility can do, but actually, any moral system in the widest acception can, not only those where an order of "Good" was distinguished. On the other hand, finding a lot of similarities in somehow (or completely) impertinent subjects (such as gory "implementation details") doesn't imply an interesting common background; finding a few similarities on pertinent subjects might not be sufficient to imply an interesting common background either. (technical remark: given a digital encoding of things, quantifying the level of interest of a common background might be expressed in terms of conditional Kolmogorov complexity.)

If we consider humans in the World, can we find cells that are "exactly the same" on distinct humans ? No, and even if we could find exactly the same cell on two humans, it wouldn't be meaningful, just boring. Yet you can nonetheless say that those two humans share the same *ideas*, the same *languages* and *idioms* or *colloquialisms*, the same *manners*, the same *Cultural Background.* And *this* is meaningful, because these are used to communicate, and greatly affect the flow of information, etc. Genetical strangers who were bred together share more background as regards society than genetical clones (twins) who were separated after their being born.

It's the same with computers: computers of the same hardware model, having large portions of common implementation code, but running completely different "applications" that have nothing conceptually common to the human user, might be considered as sharing little common information; on the contrary, even though computers may be of completely different models, of completely different hardware and software technologies, thus sharing no actual hardware or software implementation, they may still share a common background, that enables them to communicate and understand each other, and react similarly to similar environments, so that to the human users, they behave similarly, manipulate the same abstraction. *That* we called the *Operating System.*

2.10 Controversy about the Definition for an OS

When words are unfit, speech is unadapted and actions are unsuccessful

– Confucius

There have always been many lengthy arguments everytime someone proposed any definition for what an Operating System is. Many will object to the above definition of an OS, because it doesn't fit the idea they believe they had of what an OS is. Now, let's see the requirement for such a definition: as a definition, it should statically reference a consistent concept, independently enough from space, time, and current state of society and technology, so as to enable discourse about OSes in general; as applying to an existing term, it should formalize the intuitive idea generally vehiculated by this term, and as much as possible coincide with its common usage, while staying consistent (of course, where common usage is inconsistent, the definition cannot stick to the whole of it).

The definition the from previous chapter does fulfill these requirements, and it is the only one known to date by the author that fulfills them. This definition correctly identifies all the programs and user interface of Unix, DOS, Windows*, or Macintosh machines to be their respective OS, the class of similar machines being considered at each time, because they are what the user/programmer can expect to have when encountering such machines. It does support both the points of view that such software or feature, is an OS or part of the OS, or that it is not, depending on the set of machines being considered.

By "Operating System", people intuitively mean the "basic" software available on a computer, upon which the rest is built.

The first naive definition for an OS would thus be to define it by "whatever software is available with the computer when you purchase it". Now, while this sure unambiguously defines an OS, the according pertinency is very poor, because, by being purely factual, the definition induces no possible moral statement upon OSes: anything that's delivered is an OS, whatever it is. You could equivalently write some very large and sophisticated software that works, or some tiny bit of software that doesn't, still it'd be OS, by the mere fact it is sold with the computer; one could buy a computer, remove whatever was given with it, or bundle completely different packages to it, then resell it, and whatever he resells it with would be an OS. This definition, while it embodies some wisdom about the fact that the concept of OS should capture the features of actually deployed software, is so poor as to be unusable, because it isn't based on a relevant notion of deployment.

Then, one could decide that because this question of knowing what an OS is is so difficult, it should be let to the high-priests of OSdom, and that whatever is labelled "OS" by their acknowledged authority should be accepted as such, while what isn't should be deemed with the utmost defiance. While this puts the problem back, this is still basically the same attitude of accepting fact for reason, with the little enhancement that the rule of force applies to settle the fact, instead of raw facts being blindly accepted. This is abdicating reason in favor of religion. Now, the high-priests of computing that are to give a definition for an OS are not more endowed than the common computer user to give a good definition. Either they only abuse their authority to give unbacked arbitrary definitions, or they have some reasonable argument to back their definition. Since we're studying computer science, not computer religion, we can but contemptuously ignore them in the first case, and focus on their arguments in the second case. In any case, such definition by authority is useless to us.

Those who escaped the above traps, or the high-priests of the second trap, will need other criteria to define an OS. They might most obviously try to define an OS as a piece of software that does provide such and such services, to the exclusion of any other services, each taking the list of provided services from their favorite OS or OS-to-be. Unhappily, because different people and groups of people have different needs and history, they would favor differently featured OSes. Hence, they would all define an OS differently, and every such definition would disqualify every past, present and future systems, but the few ones considered from being "OSes". Hence, this conception leads to endless fights about what should or not be included in a piece of software for it to be an OS. When human civilization rather than just computer background was concerned, these would be wars and killings, crusades, colonizations and missions, in the hope to settle the one civilization over barbarism. Even without fights, we see that completely different sets of services equally qualify as OSes, much like completely different civilizations like the ancient Greek and ancient Chinese civilizations, while being completely different, both qualify as civilizations, not talking about other more or less primitive or sophisticated civilizations. Such a definition for an OS cannot be universal in time and space, and only the use force can have one prevail, so it becomes a new religion. Again, this is a poor definition for an OS.

The final step, as presented in the preceding chapter, is to define an OS as the container, instead of defining it as the contents, of the commonly available computer services; in other words, we give an intentional definition for an OS, instead of looking for an extensional definition. We saw that OS was to Computing Systems what Civilization was to Mankind; actually Computing Systems being a part of the Human system, their OSes are the mark of Human Civilization upon Computers. The language, habits, customs, scriptures, of some people, eating with one's bare hands, a fork and knife, or chopsticks, don't define *whether* these people have a civilization or not; they define *what* their civilization is. Similarly the services uniformly provided by a collection of computers, the fact that a mouse or a microphone be used as an input device, that a video screen or a braille table be used as an output device, that there be a built-in real-time clock or not, those features don't define whether those computers have an OS or not, but rather they define what is this OS they have[2].

Our definition allows us to acquire knowledge, while refusing to endorse any dogma about what we can't know; this is the very principle of information against noise, of science against metaphysics. It separates the contingencies of life from the universal concept of an OS. An OS is the common background between the computers of a considered collection. This moves the question of knowing what should or not be in an OS from a brute fight between OS religions, from the mutual destruction of dogmas, to a moral competition between OSes, to the collective construction of information. That's why we claim that our definition is more pertinent than the other ones, hence more useful, by an argument previously explained.

2.11 Operating System Utility

In an external environment which constantly changes and in which consequently some individuals will always be discovering new facts, and where we want them to make use of this new knowledge, it is clearly impossible to protect all expectations. It would decrease rather than increase certainty if the individuals were prevented from adjusting their plans of action to the new facts whenever they became known to them. In fact, many of our expectations can be fulfilled only because others constantly alter their plans in the light of new knowledge. If all our expectations concerning the actions of particular other persons were protected, all those adjustments to which we owe it that in constantly changing circumstances someboy can provide for us what we expect would be prevented. Which expectations ought to be protected must therefore depend on how we can maximize the fulfilment of expectations as a whole.

– F.A. Hayek, *Law, Legislation and Liberty*, I.4.e [4]

Let it be clear that the concept of Operating System does not apply pertinently to machines that do not evolve, that do not communicate with other machines, that do not interact with humans.

Such machines need complete, perfect, highly-optimized stand-alone software, adapted just to the specific task they are meant to accomplish. Whatever can be found in common among many such machines isn't meaningful to running those machines, as this does not influence the way information flows in the system.

However, as soon as we consider further possible versions of a "same" piece of software, as soon as we consider its incomplete development and maintenance process, the way it interacts with other pieces of software, whether in a direct or remote fashion, as soon as it has any influence on other software, be it through the medium of humans who are examining it before to build the other pieces software (or while building these), then we are indeed talking about flow of information, and the concept of OS does become meaningful.

[2]I've received reproaches about my definition including in an OS all the "interactive" parts. Firstly, my definition of an OS being formal, I wouldn't like a fuzzy concept like informal "interactivity" to be used in it. If that's to mean anything that the user can directly see on screen, then there are lots of OSes based on dynamic languages, like LISP, FORTH or RPL, where just everything is thus interactive, so a definition for an OS definitely should include such interactive things. Now, if "interactive" is to mean "purely interactive", or "not program-accessible", that is, "anything that no program written over the OS can access/modify/simulate" (which would amount to nothing in a "Good" OS, by the way), then an OS also should include interactive things, to account for all the expectations one has about the system behaving in such a way when such thing is done (typing such thing on the console, clicking with a mouse, etc). Such behavior is rightfully described in books teaching how to use such OS. So I see no reason why to exclude these from my definition of an OS. Surely, I reckon that the concepts of being user-accessible or program-accessible, are indeed interesting ones. But they are orthogonal concepts to me to what is to be expected from a random computer extracted from a considered set. Surely the conjunction of these interesting concepts might also be interesting, but these concepts are more expressive (hence more useful) when kept orthogonal. Else, how would you name the purely interactive part of what I call an OS?

See that communication between machines does not always mean that some kind of cable or radio waves be used to transmit exact messages; rather, the most used medium for machines to communicate pertinent have always been humans, those same humans who talk to each other, read each other's books, articles, and messages, then try to express some of their resulting ideas on machines.

Particularly in the case above of lots of similar perfect machines, the concept of an OS on those machines might have been meaningless, or strictly limited to a vague or limited common interface that they may offer to customers; but the concept of an OS was quite meaningful on the *development platforms* for these, where a lot of common information is potentially shared by many developers working more or less independently.

As we saw that the pertinency of a concept is related to the utility of the described object, we find the the utility of an OS lies in its dynamic aspects. An obvious dynamic aspect of the OS is how it itself evolves; but from the point of view of arbitrary user subsystems, the fundamental dynamic aspect of the OS, that dictates its Utility, is its propensity to ease communication of knowledge between the considered subsystems. Of course, these two aspects of course interact with each other.

An OS eases communication of knowledge in that it will allow to pass more Information, by providing fast broad communication lanes and information stores, but also in that it gives pertinency to this Information, thus transforming it into Knowledge, by providing a context in which to interpret received information as unambiguously as possible, and in which to synthetize new information that represent as accurately as possible the ideas that are originally meant. Note that both Quantity and Quality of Information are being considered here, and that interaction goes in both ways.

An OS will usefully evolve when modifications to a same OS project will allow improvements in the above communication of knowledge. For obvious reasons of information stability, the OS, can only evolve slowlier than its user base, and its design, which is the essence of the OS, and what manages the pertinency of Information, must change slowlier than its implementation (that drives raw performance).

2.12 Operating System Expressiveness

An Operating System is the common context in which information is passed across the Computing System. It is the one reference used for arbitrary subsystems to communicate information.

Hence, the OS dictates not so much the *amount* of information that can be passed, which is mostly limited by the laws of physics and technological hardware achievements, as it dictates the *kind* of information that can be passed, which is a question of OS design proper.

All OSes are more or less equal before technology, which is an external limitation; not all are equal before design, which is a internal limitation.

For instance, given equivalent post office administrations, two countries can ensure similarly fast and reliable shipping of goods. However, the actual use of the post office for exchanging goods will greatly depend on what warranties the state will give to both people who send and people who receive goods: how well identified are the parties, how agreements happen, how contracts are signed, how signed contracts bind the contractors, how payment happens, how disagreements are settled, how well the sent goods are guaranteed to match the advertisements, how much information people have on competing solutions, how likely a failure is likely to be, what support is available in case of failure, what recourse have parties against breach of contract by the other party, etc.

Depending on the rules followed by the system, which are part of the OS design (according to our definition of an OS), the same underlying hardware can be either an efficient way to market goods, or an inefficient risky gadget.

The Internet is a perfect example of a media with a great hardware potential for information passing, but a (currently) poor software infrastructure, that needs lots of enhancements before it can safely be used for large-scale everyday transactions.

This will surely happen, but if things go as can be predicted, there is a wide margin for improvements.

The key concept here is the *expressiveness* of the OS, which decides what kind of information is expressible by the OS.

The common misconception about expressiveness is that Turing-equivalence be all there is to it. The theory says that "all (good enough) computing systems are Turing-equivalent", in that a good enough system can simulate any other system through an simulating interpreter or simulator, so it suffices to provide a Turing-equivalent system to be able to simulate any other system. But a simulation of something is **not** the original thing. Just like the idea of money is **not** actual money[3]. The mere idea that someone may have signed a contract is not a binding contract in itself. Even the fact of actually signing a contract is not binding, in absence of any (explicit or implicit) legal context. If the system won't enforce your contracts, no one will. In absence of system support, the only enforceable contracts you can build are those where it suffices to dynamically check compliance from cooperative third-parties, and it is always legit for a party to fail. The catch is that a simulator gives **no warranty**: the meaningfulness of the result depend on the objects being manipulated respecting conventions for the validity of simulated representations. If the associated warranties can be *interned* by the very original system, then indeed that system can *express* rather than merely *simulate* the other system. If these warranties cannot be interned, then an external agent (most likely, the programmer) will have to enforce them, and you have to trust him not to fail, without recourse. Arguably, being satisfied with entering a simulation to enforce the warranties that one requires from the system is not using the original system, but building a new system above the first one, and then limiting interactions with other subsystems within that new more expressive system, while praying helplessly that that no agent in the original system should break the rules of the new system. The keyword here being "helplessly".

Some will suggest paranoidly testing for dynamic compliance of every single operation for which contracts were passed; but such an approach not only is very expensive when even feasible but is **not** a solution (though it might be better than nothing): run-time checking can detect failure to comply but it cannot enforce compliance. In everyday life, it might mean that whenever you provide a service to a stranger, this stranger may run away and not pay you back, and you have strictly no recourse, no possibility to sue or anything. At times, run-time checking may allow to take appropriate counter-measures, but it might be too late. In the case of a spacerocket (e.g. Ariane V), a runtime failure means billions of dollars that explode. In the case of a runtime failure in control software for a nuclear device (civilian reactor or military missile), I just don't dare imagine what it might mean! In any case, having some paranoid test code that will terminate the program with message "ERROR 10043: the nuclear plant is just going to unexpectedly disintegrate." won't quite help. Finally, the ability to (counter-)strike, in absence of any system control, brings new dangers that in turn meet lack of solution, as malevolent agents may strike at will.

All in all, the expressiveness of an operating system is its ability to require and provide trust, to enable exchange of trusted services, above that which can be built from zero by iterative interaction between agents. Of course, ultimately, this trust will have to rely on external, human processes. The question is how much the system relieves us

EVERYTHING BELOW IS A DRAFT, AND MUST BE COMPLETED OR REWRITTEN.....

2.13 Computing System Structure

Up to now, we've seen and discussed the *external* constraints of an OS, what is its goal, its *why*, in the implicit larger Computing System. Now that this goal is clarified, and keeping it in mind, it's time to focus on the *internal* constraints of an OS, its structure, its *how*.

The structure of an OS is the data of its characteristic components, their interrelationships, and their relationships with the rest of the computing system. We must thus study once again the structure of the whole Computing System, of which the OS is but an aspect. For this, we will once again find inspiration in considering cybernetical

[3]Please make me wrong and tell me how to convert my idea of being rich into actually being rich.

systems in general, and in comparing the situation with that of another kind of well-known cybernetical systems, human societies. The latter analogy is more than a mere metaphor, since one aspect of computing systems is as actual human societies, with users and programmers being the humans, and the running programs being activities of these humans. Indeed, until AIs come into existence, all programs are human activities (and if AIs ever exist, they won't change much of the current discourse, if we understand "human" as "sentient being").

However, the analogy is certainly not an exact correspondance (an "isomorphism"), and the way it can validly bring insight into the domain of computer systems is often more subtle than may appear at first. Most importantly, it breaks down (as far as go specific properties not true of any cybernetical system) when we consider the computerized part of computing systems, that is, programs. Indeed, computing systems comprise as basic identifiable agents not only complex unformalizable humans, but also running programs, that are quite different, simple and formalizable, and are the center of interest and of information processing in the system.

There is still some relevance to the division of computer activities depending on the human persons who run these activities, provide them with input and use their output; issues of trust between humans under the multiple roles they assume are essential in the structure of systems: when two humans or roles do not trust each other, they must rely on some kind of physical or logical separation enforced by a trusted combination of hardware, software and wetware. The ability to express and correctly implement such separation in a way that users can trust is an essential feature of an OS; When such concerns are not satisfyingly tackled by the software part of an OS, they will be tackled by the hardware part, which means buying, deploying and configuring more computers, one per user or assumed role, or by the wetware part, by having it the human users' responsibility to never do anything wrong with the capabilities with which the software entrusts them whereas it shouldn't.

But while this trust aspect covers any activity involving dynamic interaction with the external world in any cybernetical system, there is a peculiarity of computing systems that OS design can and must take advantage of: the actual data and programs that are stored in a computer are purely *extensional* entities; that is, their digital description suffice to deduce all there is to them.

extensional vs intentional aspects of programming. to take advantage of it, must respect the expected intentional modifications. Managing the coherency between diverging intentions.

Cybernetical systems

of which computing systems are a projection. Human societies are made of lots of people, each with its own needs and capabilities, desires and will. Computer societies are similarly made of these same people, considered through the limited scope of the way they interact with computers, through computers, about computers, and of the computers themselves.

People communicate with each other, and are dynamically organized in families, friendships, associations, companies, countries, confederations; every group is more or less stable;

User services vs kernel services. Privatization vs nationalization of services. Rule of Law vs State Management.

A computing system IS a human society! The programmers are the humans; the programs are their extensions.

2.14 Users *are* Programmers

To program: to influence the future behavior of the system.

Intentional vs extensional definitions.

Continuum between "Beginner" programmer vs "Advanced programmer" vs "Programmer demigod". Some will never program. Some will stay rookies forever. Some will develop good programming skills in very specific domains. Etc.

2.15 The Long Reach of the Programmer

Two scales in a computing system. Macro-scale: human minds; heuristic, evolutive. Micro-scale: automated computer programs; algorithmic, constructive.

eager stratification

Artificial barriers due to proprietary software. See other article MPFAS.

Historical barriers due to low resource availability at the time systems were designed: low-level systems.

universal system vs glue languages

managing complexity vs multiplying services

more than one way to do things? ultimately the same

2.16 Authority

Most complex enough systems are structured around a kernel, with "system services" on the one hand, and "user applications" on the other. This centralized structuration must thus be some kind of a natural concept. But why? And what are the natural attributes of a kernel? What can or cannot, must or must not, be performed by users or by the kernel ?

The principal characteristic of kernels is authority. The authority to take effective decisions that affect unwilling actors. A system programmed by a single man, by a tight team, or more generally by a one coordinated entity, doesn't need a separation of computing systems between system and user spaces; it can be just a project for a conceptually "monolithic" computer system. The necessity of a well-separated "kernel" appears as multiple people, who are not otherwise much coordinated, need to cooperate.

Monitor, Runtime, Compiler, Verifier, Trust Broker.

3 Languages and Expressiveness

3.1 Computer Languages

Firstly, let's settle what we call a "computer language".

A language is just *any* means by which humans, computers, or any active member of a dynamical system, can communicate information. Computer languages are languages used to vehiculate information about what the computer should do; any media for exchanging information is a language.

Now, this makes a language even out of point-and-click-on-window-system, or out of a bitstream protocol.

So what? Why has a language got to use ASCII symbols or a written alphabet at all? People use sounds, the deaf use their hands, various animals use a lot of different ways to communicate, computers use electrical signals. What makes the language is the structure of the information communicated, not the media used for this communication to happen.

Written or spoken english, though they have differences, are both english, and recognizable as such; what makes english is its structures, its patterns, not the media used to communicate those patterns. These patterns might be represented by things as physically foreign to each other as vibrations of the air (when one talks), or digital electrical signals on a silicon chip (when your computer text such as this very article you're reading).

Of course, symbol-based languages are simpler to implement on today's computers, but that's only a historical dependency, that may evolve and eventually disappear.

And of course not all languages are equivalent. Surely the language used to communicate with a washing machine is much more limited than what we use to talk to humans. Still, there is no reason why not to call it a language.

As with Operating Systems, the problem is not to define the concept of a computer language, but to identify what characteristics it should have to maximize its utility.

So what kind of structure shall a computer language have? What makes a language structure better or more powerful than another? That's what we'll have to inspect.

3.2 Goal of a computer language

[Rename that to "Computer Language Utility"?]

It should stressed that computer languages have nothing to do with finished, static "perfect" computer programs: those can have been written in any language, preferably a portable one (for instance, any ANSI supported language, i.e. most probably the largely supported "C", even if I'd then personally prefer FORTH or Scheme). If all interesting things already had been said and understood, and all ever needed programs already run satisfactorily on current machines, there would be no more need for a language; but there are infinitely many interesting things, and only finitely many things said and understood, so a language will always be needed, and no finite language (a grammarless dictionary) will ever be enough.

Much like an operating system, being useful not as a *static* library, but as a frame for *dynamic* computing, computer languages have to do with programming, with modifyings programs, creating new programs, not just watching existing ones; that is, computer languages are for communicating, be it with other people or a further self. That is languages are protocols to store and retrieve documents *in such a way that the meaning of a document, its dynamical properties, its propension towards evolution and modification, etc, be preserved.*

Thus, the qualities of a (programming) language do not lie only in what can eventually be done as a static program with the language; or more precisely, assuming we have all the needed "library" routines to access the hardware we need, all Turing-equivalent languages are equally able to describe any static program. These qualities do not lie in the efficiency of a straightforward implementation either, as a good "optimizing" compiler can always be achieved later, and speed critical routines can be included in libraries (i.e. if you really need a language, then you won't be a beginner for a long time at this language).

The qualities of a language lie in the easiness to express *new* concepts, and to *modify* existing routines.

With this in mind, a programming language is better than another if it is easier for a human to write a new program or to modify an existing program, or of course to reuse existing code (which is some extension to modifying code); a language is better, if sentences of equal meaning are shorter, or if just if better accuracy is reachable.

3.3 Reuse versus Rewrite

We evaluated a computing system's utility by the actual time saved by using them on the long run, as compared to using other tools instead, or not using any. Now, personal expediency suggests that people keep using the same tools as they always did, however bad they may be, and add functionalities as they are needed, because learning and installing new tools is costly. But this leads to obsolete tools grown with bogus bulging features, that provide tremendous debugging and maintenance costs. It results in completely insecure software, so no one trusts any one else's software, and no one wants to reuse other people's software, all the more if one has to pay.

For the problem is that, with existing tools, 99.99% of programming time throughout the world is spent doing again and again the same basic things that have been done hundreds of times before or elsewhere. It is common to hear (or read) that most programmers spend their time reinventing the wheel, or desesperately trying to adapt existing wheels to their gear. Of course, you can't escape asking students and newbies to repeat and learn what their elders did, so they can understand it and interiorize the constraints of computing. The problem is that today's crippled tools and closed development strategies make learning difficult and reuse even more difficult, secure reuse being just impossible. Thus people spend most of their time writing again and again new versions of earlier works, nothing really worth the time they spend, nothing *original*, only so they can be sure they know what it does, and it provides correctly the particular feature they need that couldn't be done before, or at least not exactly. Even then, they seldom manage to have it do what they want.

Now, after all, you may argue that such a situation creates jobs, so is desirable; so why bother ?

First of all, there is plenty of useful work to do on Earth, so time and money saved by not repeating things while programming can be spent on many many other activities (if you really can't find any, call me, I'll show you). Physical resources are globally limited, so wasting them at doing redundant work is unacceptably harmful.

Paying people to dig holes and fill them back just to create jobs, as suggested by despicable economists like J.M. Keynes, is of utmost stupidity. Else, we might encourage random killing, as it decreases unemployment among potential victims, and increases employment among morticians, cops, and journalists. If Maynard Keynes' argument holds, I particularly recommend suicide to its proponents for the beneficial effect it has on society. See Bastiat's works [1] for a refutation of this myth, more than a hundred years before stupid socialist politicians apply it: maybe spending money to do useless things might have some beneficial aspects, as for example stimulating employment; but their global effect is very harmful, as the money and energy spent by central organs to the limited benefit of a few could have been spent much more usefully for everyone (not forcibly by a central organ *at all*), as there are so many useful things to be done, be it only to prepare against natural catastrophes, not to talk about human curses. That useless work policy is taking a lot from everyone to give little to a few.

Now, rewriting is a serious problem for everyone. To begin with, rewriting is a loss of time, that make programming delays quite longer, thus is very costly. More costly even is the fact that rewriting is an error prone operation and anytime during a rewrite, one may introduce errors very difficult to trace and remove (if need be, one may recall the consequences of computer failures in space ships, phone nets, planes). Reuse of existing data accross software rewrites, and communication of data between different software proves being of exorbitant cost. The most costly aspect of rewriting may also be the fact that any work has a short lifespan, and will have to be rewritten entirely from scratch whenever a new problem arises; thus programming investment cost is high, and software maintenance is of high cost and low quality. And it is to be considered that rewriting is an ungrateful work that disheartens programmers, which has an immeasurably negative effect on programmer productivity and work quality, while wasting their (programming or other) talents. Last but not least, having to rewrite from scratch creates an limit to software quality, that is, no software can be better than what one man can program during one life.

Rewrite *is* waste of shared resources by lack of communication. And all the argument is about that: not communicating is harmful; any good the system should encourage communication. Now, even when current operating systems greatly limit communication of computer code, they happily do not prevent humans to communicate informal ideas of computer code. This is how we could get where we are.

Therefore, it will now be assumed as proven that code rewriting is a really bad thing, and that we thus want the opposite: software *reuse*, software *sharing*.

We could have arrived at the same conclusion just with this simple argument: if some software is really useful (considering the general interest), then it must be used many, many times, by many different people, unless it is some kind of computation with a definitive answer that concerns everybody (which is difficult to conceive: some software that would solve a metaphysical or historical problem!). Thus, useful software, least it be some kind of very unique code, is to be reused countless times. That's why to be useful, code must be very easy to reuse.

It will be showed that such reuse is what the "Object-Orientation" slogan is all about, and what it really means when it means anything. But reuse itself introduces new problems that have to be solved before reuse can actually be possible, problems as we already saw, of *trust*: how can one trust software from someone else? How can one reuse software without spreading errors from reused software, without introducing errors due to misunderstanding or misadaptation of old code, and without having software obsolescence? We'll see what are possible reuse techniques, and how they cope with these problems.

3.4 Copying Code

The first and the simplest way to reuse code is just the "copy-paste" method: the human user just copies some piece of code, and pastes it in a new context, then modifies it to fit a new particular purpose.

This is really like copying whole chapters of a book, and changing a names to have it fit a new context; this method has got many flaws and lacks, and we can both moral and economically object to it.

First of all, copying is a tedious and thus error-prone method: if you have to copy and modify the same piece of code thousands of times, it can prove a long and difficult work, and nothing will prevent you from doing as many mistakes while copying or modifying.

As for the moral or economical objection, it is sometimes considered bad manners to copy other people's code, especially when copyright issues are involved; sometimes code is protected in such a way that one cannot copy it easily (or would be sued for doing that); thus this copy-paste method won't even be legally of humanly possible everytime.

Then, assuming that the previous problems could be solved (which is not obvious at all), there would still be a *big* problem about code copying: uncontrolled propagation of bugs and lacks of feature accross the system. And this is quite a serious threat to anything like code maintenance; actually, copying code means that any misfeature in the code is copied altogether with intended code. So the paradox of code copying is that bad copying introduces new errors, while good copying spreads existing errors; in any case code copying is an error prone method. Error correction itself is made very difficult, because every copy of the code must be corrected according to its own particular context, while tracking down all existing copies is especially difficult as code will have been modified (else the copy would have been made useless by any macro-defining preprocessor or procedure call in any language). Moreover, if another programmer (or the same programmer some time later) ever wants to modify the code, he may be unable to find all the modified copies.

To conclude, software creation and maintenance is made very difficult, and even impossible, when using copy-paste; thus, this method is definitely bad for anything but exceptional reuse of a small number of mostly identical code in a context where expediency is much more important than long-term utility. That is, copy-paste is good for "hacking" small programs for immediate use; but it's definitely not a method to program code meant to last or to be widely used.

3.5 Having an Extended Vocabulary...

The second easiest, and most common way to reuse code, is to rely on *standard libraries*. Computer libraries are more like dictionaries and technical references than libraries, but the name stuck. So places where one can find lots of such "libraries" are called repositories.

Using a standard library is easy: look for what you need in the standard library's index, carefully read the manual for the standard code you use, and be sure to follow the instructions.

Unhappily, not everything one needs will be part of a standard library, for standard library include only things that have been established as needed by a large number of persons. Patiently waiting for the functionality one needs to be included in a next version of standard libraries is not a solution, either, because what makes some work useful is precisely what hasn't been done before, so that even if by chance the functionality gets added, it would mean someone else did the useful work in one's place,

..... not everything there are good reasons why before a standard library is available You wait for the function you need to be included in the standard library, and then use it as the manual describes it when it is finally provided.

standards are long to come, and are even longer to be implemented the way they are documented. By that time, you will have needed new not-yet-standard features, and will have had to implement them or to use non-standard dictionaries; when the standard eventually includes your feature, you'll finally have to choose between keeping a non-standard program, that won't be able to communicate with newer packages, or rewriting your program to conform to the standard.

Moreover, this reuse method relies heavily on a central agency for editing revised versions of the standard library. And how could a centralized agency do all the work for everyone to be happy ? Trying to impose reliance on a sole central agency that is communism. Relying only on multiple concurrent hierarchically organized agencies is feudalism. Oneself is the only thing one can ultimately rely upon; and liberalism tells us that only by having the freeer the information interchange between people, the better the system.

It's like vocabulary, culture: you always need people to write dictionaries, encyclopaedias, and reference textbooks; but these people just won't ever provide new knowledge and techniques, they rather settle what everyone already know, thus facilitating communication where people had to translate between several existing ones more easily. You still need other people to create new things: you just can't wait for what you need to be included in the next revision of such reference book; it won't ever be if no one does settle it clearly before it may be considered by a standardization commitee.

Now, these standard dictionaries have a technical problem: the more useful they strive to be, the larger they grow, but the larger they grow, the more difficult it gets to retrieve the right word from its meaning, which is what you want when you're writing. That's why we need some means to retrieve words from their subject, their relationship with other words; thus we need a language to talk about properties of words (perhaps the same), about how words are created, what words are or not in the standard dictionary and will or will not be. And this language will have to evolve too, so a "meta"-library will not be enough.

When vocabularies grow too large, there appear "needle in haystack" problems: though it exists, you can't locate the word you're looking for, because there's no better way to look for it than to cautiously read the entire dictionary until you come to it...

3.6 ... or a Better Grammar

Furthermore, how is a dictionary to be used ? A dictionary does not deal with new words; only old ones. To express non-trivial things, one must do more than just pronounce a one magic word; one must *combine* words into meaningful sentences. And this is a matter of *grammar* - the structure of the language - not *vocabulary*. We could have seen that immediately: standard libraries do not deal with writing new software, but with sharing old software, which is also useful, but comes second, as there must be software before there can be old software. Computer software was not created, but develops from a long tradition. So a library is great for reuse, but actually, a good grammar is essential to use itself, and reuse in particular.

That is, the right thing is not *statically* having a extended vocabulary, but *dynamically* having an extended vocabulary; however statically extended, the vocabulary will never be large enough. Thus we need good dynamical way to define new vocabulary. Again, it's a matter of dynamism versus statism. Current OSes suck because of their statism. Dynamically having an extended vocabulary means having dynamic ways to extend the vocabulary, which is a matter of grammar, not dictionary.

Now what does reuse mean for the language grammar ? It means that you can define new words from existing ones, thus creating new contexts, in which you can talk more easily about your particular problems. That is, you must be able to add new words and provide new, extended, dictionaries. To allow the most powerful communication, the language should provide all meaningful means to create new words. To allow multiple people whose vocabularies evolve independently to communicate their *ideas*, it should allow easy abstraction and manipulation of the context, so that people with different context backgrounds can understand exactly each other's ideas.

Thus we have two basic constructions, that shall be universally available: extracting an object's value in a context (commonly called beta-reduction), and abstracting the context of an object (commonly called lambda-abstraction). A context is made of variables. When you reduce an object, you replace occurences of the variable by its bound value; when abstracting the context, you create an object with occurences of an unbound variable inside, that you may reduce later after having bound the variable. We thus have a third operation, namely function evaluation, that binds an object to a free variable in a context.

For the grammar to allow maximal reuse, just any object shall be abstractible. But what are those objects ?

3.7 Abstraction

.....

The theory of abstractions is called lambda-calculus. There are infinitely many different lambda-calculi, each having its own properties.

Basically, you start with a universe of base objects. .... Base objects, or zero-order objects... first order ... second order ... nth order ... higher order ... reflectivity ... beware of reflectivity of a sub-language, not the language itself ... syntax control ... .

.....

(genericity ?)

.....

3.8 Metaprogramming

.....

3.9 Reflection

.....

3.10 Security

We already saw how the one big problem about reusing software is it that when you share the software, you share its good features, but you also share its bugs.

Reuse is good when it saves work, but you can't call that saving work when it makes you spend so much more time tracking bugs, avoiding them, fearing them, trying to prevent their effects, that you would have been better rewriting the software from scratch so you could trust it.

That's why sharable software is useless if it is not also trustworthy software.

Firstly, we must note that this worry about security does not come from software sharing; it is only multiplicated and propagated by software sharing. Even when you "share" code only with your past and future selves, the need arises. The problem is you're never sure that a module you use does what *you* expect it to. Moreover, to be sure you agree with the module, you must have some means to know what you want, and what the author intended. And this won't warranty that the module works as intended by the author. ......

The first idea that arises is then "programming by contract", that is, every time some piece of code is called, it will first check all the assumptions made on the parameters, and when it returns, the caller will check that the result does fill all the requirements. This may seem simple, but implementing such technique is quite tricky: it means that checking the parameters and results is easy to do, and that you trust the checking code anyway; it also implies that all the necessary information for proper checking is computed, which is loss of space, and that all kind of checking will take place, which is loss of time. The method is thus very costly, and what does it bring ? Well, the program will just detect failure and abort ! Sometimes aborting is ok, when you have time (and money) to call some maintenance service, but sometimes it is *not*: a plane, a train, a boat, or a spacecraft whose software fail will crash, collide, sink, explode, be lost, or whatever, and won't be able to wait for repairs before it's too late. And even when lives or billion dollars are not involved, any failure can be very costly, at least for the victim, who may be unable to work. That's why security *is* something important that any operating system should offer support for. Why integrate such support in the OS itself, and not on "higher layers" ? For the very same reasons that reuse had to be integrated to the OS: because else, you would have to use not the system, but a system built on top of it, with all the related problems, and you would have to rely on the double (or bigger multiple, in case of multiple intermediate layers) implementations, that each introduce unsecurity (perhaps even bugs), unadapted semantics, big loss in performance.

......

3.11 Trusting programs

So we just saw techniques to design trustworthy software. Now, how could you be sure they were well used (if at all), unless you did participate to the design using them ? These techniques can only enforce trust to the technician people who have access to the internals of the software. What kind of trust can the user expect from some software s/he purchased ?

Some companies sell support for software, so they shall repair or replace computer systems in case the customer may have problems. Support is fine indeed; support is even needed by anyone seriously using a computer (now which kind of support, it depends on what the customer needs, and what he can afford). But support won't ever replace reliable software. You never can repair all the harm that may result from misdesigned software when used in critical environment: exploding spacecrafts, shutdown phone networks, missed surgical operation, miscomputed bank accounts, blocked factories, all these cost so much that no one can ever pay back. Thus, however important, the computer support one gets is independent from the trustworth of the software one uses.

The computer industry offers no guarantee to its software's reliability. You have to trust them, to trust their programmers and their sellers. But you shouldn't, as their interest is to spend as few money as possible in making their software reliable, as long as you buy it. They may have some ethics that will bind them to design software as reliable as they can; but don't count on ethics to last indefinitely, when there is. The only way to make sure they strive is to have some pressure on them, so that in case they would cheat you, you threaten software vendors to sue them (and long when even possible), or to lead a campaign against buying their products.

The former very hard, when possible at all, and last for years during the which you must feed lawyers, be worried, without being sure to win. The latter means there is fair competition, so you can choose a product that will replace the one that fails; it also means that competing software allow to recover your data from the flawed system, and run on your former hardware. So even competition isn't enough if it's wild and uncontrolled, and vendors can create de facto monopolies on software or hardware compatibility (which they do).

The only reason why you should trust software is that everyone can, and many do, examine, use and test *freely* the software *and* its actual or potential competitors, and still keep using it. We shall insist on there being potential competitors, to which you may compare only if the software sources and internal documentation is freely available, which is open development, as compared to development with non-disclosure agreements. This is the one true heart of liberalism.

Now, what if the software you use is too specific to be used and tested by many ? What if there's no way (at reasonable price) to get feedback from the other actual and potential users of the software ? What if you don't have time to choose before you can get enough feedback to make some worthwhile opinion ? In those cases, the liberal theory above won't apply anymore.

3.12 Program proof

As the need of security in computer systems grows, one can't satisfy himself with trusting all the modules one uses, just because other people were (alledgedly) happy with them, or the authors of the modules have a good reputation, or other people bought it but there's no way to get feedback, or (silly idea) one paid a lot for it, or have been promised an "equal" replacement (but no money back for the other loss) in case it fails.

However, trusting a computer system is foremost when lives (or their contents) are involved by the correct behavior of a module.

Thus, providers of computer system modules will have to provide some reliable warranty that their modules cause no harm. They may offer to pay back any harm that may result from bugs (but such harm is seldom measurable). Or they may offer a *proof* of the correctness of their program.

Test suites are pretty, but not very significant. What are tests in a million cases, when there are multi-zillions of them, or even infinitely many ? Test suites are due to fail.

Computers were born from mathematicians and their theory is largely developped. If computer systems are designed from mathematically simple elements, that have well-known semantics, it may be actually possible to prove that the computer system actually does what it is meant to do.

The advantage of a mathematical proof is that, *when done according to the very strict rules of logic*, it is as accurate as a comprehensive test, even though such test may be impossible because the number of cases so wondrous (when not infinite) that it would take far longer than the age of the universe to check each one even at the speed of light.

Now, proving a program's correctness is a difficult task, whose complexity grows uncontrollably with the size of the program to prove. This is why the need to use computer systems arises quickly for such proof. Thus, to trust the proof, you must also trust the computer proofchecking program. But this program can be *very short* and easy to understand; it can also be made publicly available, and be examined, used, tested, by all the computer users and hackers throughout the world, as explained previously, because it is useful to everyone indeed. If those requirements are fulfilled, such program may be really much more reliable than the most reknowned human doing the same job.

Anyway, the simplest are the specifications and proofs, the most reliable they are too. Therefore, programmers ought to use the programming concepts that allow the easiest proofs, such as pure lambda-calculus, as used in a language like like ML. Any kind of thing like side-effects and shared (global) variables should be avoided whenever possible. The language syntax should remain always clear and as localized as possible. As for the efficiency hungry, we recall that however fast to execute, an unreliable program is worthless, while today's compiler technology is ready to translate mathematical abstractions into code that is almost as fast as the unreliable software obtained by the so-called "optimizing" compilers for unsafe languages.

Of course, having program proofs does not mean we should be less careful. If you misspecify what a program must do, and prove the program fulfills the bogus specification, you may have a bogus program; so you must be careful to undertand well what the program is meant to do, and how to express it to the proofchecker. Also, proofs are always founded on assumptions. With wrong assumptions, you can prove anything. So program proofs mean we should always be careful, but we may at last concentrate on the critical parts, and not lose our time verifying details, which computers do much better than us.

Actually, that's what machines, including computers, are all about: having the human concentrate on the essential, and letting the machines do repetitive tasks.

A programming language is low level when its programs require attention to the irrelevant.

– Alan Perlis

4 No Computer is an Iland

5 Conclusion

The ideas exposed in this article are not new. Since the seventeenth century, political thinkers, economists, physicists, biologists, have discovered them, which led to theories of democracy, liberalism, thermodynamics, darwinism. The unification of these into a same set of principles, under a more general theory of information, is not foreign to the appearance of computer technology, from the early essays of Leibniz on automatas, to Wiener's Cybernetics.

The point of the Tunes project is not to claim to have invented any of these, neither is it to claim to realize anything technologically original. The claim of this article is to consistently acknowledge the validity of these principles in the computer world that is so well suited to experiment them, by its very principle of manipulating information exactly.

The Tunes project will try to provide an initial software frame for reliable distributed information to exist, that is all the more needed that the necessary hardware is already available and underexploited by people following the transient external aspects of tradition instead of its stable roots.

A Draft

Nota Bene: This section contains many ideas to insert in the text as it is rewritten. The ideas are in no particular order (not even order of chronological appearance), having been put at random places in the file as they came, or were moved from the written text, since late january 1995 when redacting this article began.

A.1 About the whole article

Some of this draft should definitely be moved to other Tunes documentation files, or expanded into independent articles.

A.2 Part I

Part I would:

1. Show that OS utility lies in its influence on dynamic CS behavior

2. The OS is not as much the software as the *protocols*

3. Show that this influence is in the way the common background allows to increase signal/noise ratio, that is to give meaning to observable data, to provide expressive languages using the obsersable world as substratum

4. The role of the "kernel" is to provide some central authority as a ultimate resource to arbitrate conflicts and guarantee consistency.

5. Constraints of an Operating System:

   (a) it may contains only a tiny fraction of the total information in the CS, as its information is bounded by what one computer can know, whereas the system is bounded by what N computers can know.

(b) evolves slowly, in a conservative way so that dataflow can rely on it.

(c)

6. (old stuff)

evolution computing is a recent art whose evolution is well-known

latest multimedia is the latest OS slogan; when we see through this veil of illusion, we find

(a) [tradition] the trend is toward adding new functions to the OS. and the trend in which they evolve

(b) [u vs p] show that they fail

(c) [u vs p]

see what is their approach,

ii. see why it fails

iii. The essence of an OS is no more in a kernel that would supervise all forms of communication between objects, than the essence of civilization lies in a central administration that would supervise all forms of communication between humans. The essence of an OS is in the abstract property of allowing objects to communicate, through any possible decentralized means; it is in its utility as a general context for communication, much as civilization is an intangible set of said or unsaid traditions and rules, that allow humans to rely on each other.

(d) [multiplex] focus on what they should do, not on what they do (what defines a place setting is not its having the shape of a fork or that of a spoon, but its ability to ease lunch activity, that is, its function, not its implementation). (xref to PartII: centralize)

1. Part I:

(a) (I.10 ?)

utility – correlation to static ou dynamic features

[current OSes] informational basis that gives meaning to the flux of raw information; dynamical structure

(b) (I.11 ?) kernel, centralism, authority

(c) (I.12 ?) The ultimate source of meta(n)-information: Man

2. Security is being able to devise arbitrary contracts, and have the guarantee that if agreed upon, the contract will be fulfilled.

Systems that don't allow you to express the contract you want are stupid unsecure systems.

Systems that do allow you to express the contract you want, but have no way to enforce it (e.g. literate programming) are ineffective unsecure systems.

Systems that enforce contracts that you don't want are fascist unfree systems.

3. and only such information can eventually and enrich the whole system. basis of any reliable information upon which new information can be built that will enrich the whole system; when this information eventually settles, it enriches in turn the OS, and can serve as a universal basis for even further enhancements. That is the utility of Operating Systems.

That's why the power and long-term utility of an OS mustn't be measured according to what the OS does currently allow to do, but according to how easily it can be extended so that more and more people share more and more complex software. That is, the power of an OS is not expressed in terms of services it *statically* provide, but in terms of services it can *dynamically* manage; intelligence is expressed not in terms of knowledge, but in terms of evolutivity toward more knowledge. A culture with a deep knowledge but that would prevent or considerably slowdown further innovations, like the ancient chinese civilization, would indeed be quite harmful. An OS providing lots of services, but not allowing its user to evolve would likewise be harmful.

Utility lies in new, *original* information; a large body of acquired information is a sign of *past* utility, but quite independent from *current* utility.

Again, we find the obvious analogy with human culture for which the same stands; the analogy is not fallacious at all, as the primary goal of an operating system is allowing *humans* to communicate with computers more easily to achieve better software. So an operating system *is* a part of *human* culture, though a part that involves computers.

Multiplying the actual services provided by an operating system may be an expedient way to solve computer problems, in the same way that multiplying welfare institutions may be an expedient way to solve the everyday problems of a human system; the progress of the system ultimately means that those services will actually be multiplied in the long run. However, from the point of view of utility, what counts is not any the objective state of the system at any given moment, and its ephemeral advantages, but the dynamic project of the system across time, and its smaller, but growing, long-standing advantages.

the information in an OS is virtually (not forcibly physically) duplicated at each node. Hence growing the OS for ever more feature is harmful, as it would involve an ever increased waste of resources duplicated at each node, instead of letting each node develop original information in a way adapted to its immediate environment.

A.3 Users *are* Programmers

The only source of information in the UCS that we can directly act upon, hence what counts with respect to utility, is the Humans. Therefore, Operating Systems should structure the Computing System so that the fullest possible human creativity is promoted.

.....

The deepest flaw in computer design is this idea that there is a fundamental difference between system programming and usual programming, between usual programming and "mere" using. The previous point shows how false is this conception.

The truth is any computer user, whether a programming guru or a novice user, is somehow trying to communicate with the machine. The easier the communication, the quicker better larger the work is getting done.

Of course, there are different kinds of use; actually, there are infinitely many. You can often say that such kind of computer use is much more advanced and technical than such other; but you can never find a clear limit, and that's the important point (in mathematics, we'd say the space of kinds of computing is connected).

Of course also, any given computer object has been created by some user(s), who programmed it above a given system, and is being used by other (or the same) user(s), who program using it, above the thus enriched system. That is, there are computer object providers and consumers. But anyone can provide some objects and consume other objects; providing objects without using some is unimaginable, while using objects without providing any is pure useless waste. The global opposition between users and programmers that roots the computer industry is thus inadequate; instead, there is a local complementarity between providers and consumers of every kind of objects.

Some say that common users are too stupid to program; that's only despising them; most of them don't have *time* and *mind* to learn all the subtleties of advanced programming; Most of the time, such subtleties shouldn't be really needed, and learning them is thus a waste of time but they often do manually emulate macros, and if shown once how to do it, are very eager to use or even write their own macros or aliases.

Others fear that encouraging people to use a powerful programming language is the door open to piracy and system crash, and argue that programming languages are too complicated anyway. Well, if the language library has such security holes and cryptic syntax, then it is clearly misdesigned; and if the language doesn't allow the design of a secure, understandable library, then the language itself is misdesigned (e.g. "C"). Whatever was misdesigned, it should be redesigned, amended or replaced (as should be "C"). If you don't want people to cross an invisible line, just do not draw roads that cross the line, write understandable warning signs, then hire an army of guards to shoot at people trying to trespass or walk out of the road. If you're really paranoid, then just don't let people near the line: don't have them use your computer. But if they have to use your computer, then make the line appear, and abandon these ill-traced roads and fascist behavior.

So as for those who despise higher-order and user-customizability, I shall repeat that there is *NO* frontier between using and programming. Programming *is* using the computer while using a computer *is* programming it. Which does not mean there is no difference between various users-programmers; but creating an arbitrary division in software between "languages" for "programmers" and "interfaces" for mere "users" is asking reality to comply to one's sentences instead of having one's sentences reflect reality: one ends with plenty of unadapted, inefficient, unpowerful tools, stupefies all computer users with a lot of unuseful ill-conceived, similar but different languages, and wastes a considerable lot of human and computer resources, writing the same elementary software again and again.

A.4 Operating System Kernel

In traditional OS design, the kernel is some central piece of software through which any communication between first-class system objects is done...

But this accounts only for centralized design; it appears that what system acknowledge as first-class objects are actually very coarse-grained information concepts, and that a meaningful study of information flow should take into account much finer-grained information, that such system just do no consider at all, hence being unadapted to the actual use that is done of them.

How does this design generalize to arbitrary OSes? What do OS kernels provide that is essential to all OSes, and what do they do that is costly noise?

To answer such questions, we must depart from the traditional OS point of view that we know is flawed, and see how are OSes doing, that we recognized as such, that traditional design refuses to consider this way, and what the analogy to human systems lead to.

Thus, we see that of course, centralization of the information flow through the kernel is not needed: hence, information most often is much more efficiently passed directly from object to object without any intermediate. Also,

To conclude, we'll say that the kernel is the central authority used to coordinate software components, and solve conflicts, in a computer system.

A.5 Current state of System software

It is remarkable that while since their origins, computer hardware have grown in power and speed at a constant exponential rate, system software only slowly evolved in comparison. It does not offer any new tools to master the increasing power of hardware, but only enhancements of obsolete tools, and new "device drivers" to access new kinds of hardware as they appear. System software becomes fatware (a.k.a. hugeware), as it tries to cope differently with all the different users' different but similar problems.

It is also remarkable that while new standard libraries arise, they do not lead to reduced code size for programs of same functionality, but to enhanced code size for them, so that they take into account all the newly added capabilities.

As a blatant example of the lack of evolution of system software quality is the fact that the most popular system software in the world (MS-DOS) is a fifteen-year old thing that does not allow the user to do either simple tasks, or complicated ones, thus being a no-operating system, and forces programmers to rewrite low-level tasks everytime they develop any non-trivial program, while not even providing trivial programs.

This industry-standard has always been designed as a least sub-system possible for the Unix system, which itself is a least subsystem of Multics made of features assembled in undue ways on top of only two basic abstractions, the raw sequence of bytes ("files"), and the ASCII character string.

As these abstractions proved not enough to express adequately the semantics of new hardware and software that appeared, Unix has had a huge number of ad-hoc "system calls" added, to extend the operating system in special ways. Hence, what was an OS meant to fit the tiny memory of then available computers, has grown into a tentaculous monster with ever growing pseudopods, that wastes without counting the resources of the most powerful workstations. And *this*, renamed as POSIX, is the new industry standard OS to come, whose promoters crown as the traditional, if not natural, way to organize computations.

Following the same tendency, widespread OSes are found upon a large number of human interface services, video and sound. This is known as the "multi-media" revolution, which basically just means that your computer produces high-quality graphics and sound. All that is fine: it means that your system software grants you access to your actual hardware, which is the least it can do!

But software design, a.k.a. programming, is not made simpler for that; it is even made quite harder: while a lot of new primitives are made available, no new combinatorials are provided that could ease their manipulation; worse, even the old reliable software is made obsolete by the new interface conventions. Thus you have computers with beautiful interfaces that waste lots of resources, but that cannot do anything new; to actually do interesting things, you must constantly rewrite everything from almost scratch, which leads to very expensive low-quality slowly-evolving software.

A.6 An Ancien Régime

[= most of the energy is wasted in a fight for supremacy between monopolies]

The current computing world is anything but a failure. So many things are now done by computers that relieve people from stupid repetitive work, and so many things are done that just could not be done without computers, that nobody can deny the utility of today's computers relatively to the implicit reference being the absence of computers.

But somehow, programming techniques are finding their limits as programs reach the size beyond which no human can fully understand the whole of one. And the current OS trend, by generating code bloat, makes those limits reached much faster than they should, while wasting lots of human resources. It is thus necessary to see why current programming techniques lead to code bloat, and how this trend can be slowed down, set back, or reversed.

Of course, we easily can diagnose about the "multimedia revolution" that it stems from the cult of external look, of the container, to the detriment of the internal being, the contents; such cult is inevitable whenever non-technical people have to choose without any objective guide among technical products, so that the most seductive wins. So this general phenomenon, which goes beyond the scope of this paper, though it does harm to the computing world, and must be fought there as well as elsewhere, is a sign that computing spreads and benefits to a large public; by its very nature, it may waste a lot of resources, but it won't compromise the general utility of operating systems. Hence, if there is some flaw to find in current OS design, it must be looked for deeper.

Computing is a recent art, and somehow, it left its Antiquity for its Ancien Régime. Its world is dominated by a few powerful companies, that wage a perpetual war to each other, where At the same time, there are heavens where computists can grow in art while freely benefitting ..... isn't the deeply rooted .....

Actually, the ..... the informational status of the computer world is quite remindful of the political status of .....

A.7 Computists

A.8 Contents of an Operating System

What are the characteristic components of an operating system ?

Well, firstly, we may like to find some underlying structure of mind in terms of which everything else would be expressed, and that we would call "kernel". Most existing OSes, at least, all those software that claim to be an OS, are conceived this way. Then, over this "kernel" that statically provides most basic services, "standard libraries" and "standard programs" are provided that should be able to do all that is needed in the system, that would contain all the system knowledge, while standard "device drivers" would provide complementary access to the external world.

We already see why such a conception may fail: it could perhaps be perfect for a finite unextensible static system, but we feel it may not be able to express a dynamically evolving system. However, a solid argument why it shouldn't be able to do so is not so obvious at first sight. The key is that like any complex enough systems, like human beings, computer have some self-knowledge. The fact becomes obvious when you see a computer being used as a development system for programs that will run on the same computer. And indeed the exceptions to that "kernel" concept are those kind of dynamic languages and systems that we call "reflective", that is, that allow dynamical manipulation of the language constructs themselves: FORTH and LISP (or Scheme) development systems, which can be at the same time editors, interpreters, debuggers and compilers, even if those functionalities are available separately, are such reflective systems. so there is no "kernel" design, but rather an integrated OS.

And then, we see that if the system is powerful enough (that is, reflective), any knowledge in the system can be applied to the system itself; any knowledge is also self-knowledge; so it can express system structure. As you discover more knowledge, you also discover more system structure, perhaps better structure than before, and certainly structure that is more efficiently represented directly than through stubborn translation to those static kernel constructs. So you can never statically settle once and for all the structure of the system without ampering the system's ability to evolve toward a better state; any structure that cannot adapt, even those you trust the most, *may* eventually (though slowly) become a burden as new meta-knowledge is available. Even if it actually *won't*, you can never be sure of it, and can expect only refutation, never confirmation of any such assumption.

The conclusion to this is that you cannot truly separate a "kernel" from a "standard library" or from "device drivers"; in a system that works properly, all have to be integrated into the single concept, the system itself as a whole. Any clear cut distinction inside the system is purely arbitrary, and harmful if not done due to strong reasons of necessity.

A.9 Toward a Unified System

From what was previously said, what can we deduce about how an OS should be behaved for real utility ?

Well, we have seen that an OS' utility is not defined in terms of static behavior, or standard library functionality; that it should be optimally designed for dynamic extensibility, that it shall provide a unified interface to all users, without enforcing arbitrary layers (or anything arbitrary at all). That is, an OS should be primarily open and rational.

But then, what kind of characteristics are these ? They are features of a computing language. We defined an OS by its observational semantics, and thus logically ended into a good OS being defined by a good way to communicate with it and have it react.

People often boast about their OS being "language independent", but what does it actually mean ? Any powerful-enough (mathematicians say universal/Turing-equivalent) computing system is able to emulate any language, so this is no valid argument. Most of the time, this brag only means that they followed no structured plan as for their OS semantics, which will lead to some horrible inconsistent interface, or voluntarily limited their software to interface with the least powerful language.

So before we can say how an OS should be, we must study computer languages, what they are meant to, how to compare them, how they should be or not.

Comparing computers and cars:

(ai) people say that computers, like cars, should have everything done by the machine, with the user never having to modify anything.

ii. but cars are rarely creative objects Most people use cars to move from some place to another, which they don't consider as a piece of art, as a work they produce. They rather feel it's some inevitable noise, that should be reduced as much as possible.

iii. cars are merely tools to relieve people from the burden of displacement, and even then, we don't forbid people from repairing their car themselves, or adding something to it, or making it. Of course, there are laws about how cars should or should not be done, that these persons should follow like all manufacturers, for security reasons.

iv. Thus, in so far as computers are tools that people are not developing, everything should be made to relieve people from the hassle of using the computer, to hide all the nasty details, to provide everything possible to make their daily computer usage easy and secure, fool-proof, etc, to the detriment of raw performance, and even of some "liberties" that bring only chaos (like the liberty to drive on either side of the road would be).

v. This is a sign that Computing as a project evolves, and the obtained computerware are objects that this project leaves behind it; the more advanced the project, the more elaborate these objects indeed.

vi. Now, information technology, under its particular form of computers as well as all of its forms, is precisely not a complete project, but a project in continuous development.

vii. Surely, people should not have to worry about completed parts, (though they should not be prevented to worry about them either).

viii. but more importantly, they should be able to freely contribute to the project.....

(b) The problem is that the society as it is does not regard meta-information as more powerful than terminal information; it tends to judge things according to their cost instead of judging them according to their value.

as if only a class of people should learn to read, and do all the work of reading in place of other People butis precisely an art that is ever-developing. It could be said that what the are not in development anymore, that become more and more objects,

(c) Defining an OS as a set of low-level abstractions: If freight technology had been left to similar companies and academies, they would have put the horse as the only basic abstraction on which to build... They could have made a new standard when it would have been obvious that steam engines should have been adopted twenty years ago, and similarly for all newer technologies... In any case, it doesn't directly tackle the real problem, which is reliable transport of goods; it just forces to people to use standard technology, however it be obsolete, and prevents them from developping structures that would survive this technology.

Also see the disaster of State managing services.

(d) Under a free programming system, independent software modules are made independently for independent purposes. Under current bound programming systems, software are not modular, not independent, and if you can't convince an established company that your purpose does match that of enough ready-to-pay people (money-wise), you just can't write it at all.

Hence Free software means that more software will be written, that it will have more feedback, hennce will be better in turn, etc.

(e) See the failure of the french industry, because of its illusory policy of developing their "own standard" (i.e. not a standard, as they're not strong enough to impose it by force) in a way bot internally centralized and externally isolated (!!!); such an industry can survive only by constantly stealing the taxpayers, and that's exactly what it has been doing for decades.

(f) Let's use the limited metaphor of the computing system as a human society:

i.
ii. at its basis is a Constitution, which has the double role of acknowledging the few informal rules that are found as universal requirements for a just society, and of settling arbitrary general settings as an agreed-upon frame in which those requirements can be provided.
iii. then are laws, bills, and conventions, that are arbitrary binding but renegociable contracts, made whenever a common solution is needed to some shared problem.
iv. then are the executives, who must do the minimal work of verifying that the legal constraints are always respected, that information does flow freely, and that noise and disinformation are discouraged.

Well, then

i. the operating system "kernel" is like State – it regulates interaction between objects;
ii. the very basics of the system are like the constitution – a set of informal rules that explains the general principles of the system and establish a common arbitration.
iii. Standard protocols are like Laws and Rules – they provide common features at the expense of common constraints, that defines the way object interact.
iv. Laws et al should include arbitrary decisions insofar as and as long as the fact that an arbitrary decision is made itself is not arbitrary: see the classical problem of everyone driving on the same side of the roads; whether everyone should drive on the right side or on the left side of the road is essentially arbitrary; that everyone should either drive on the right side or on the left side of the road is not.
v. The most common error about State and Operating systems is to believe that either should actually MANAGE the system and ultimately do or have do everything about it. That's completely WRONG. They should *regulate* things and
vi. States are the skeleton of Societies. if the skeleton was all that counted in a man, men would be ... skeletal! Surely the skeleton is important, but it is not it that will make the man move; it will only serve as a background that supports the move.
vii. As an example, the Académie Française, meant to represent France's most proeminent litterary authors, is NOT meant to write all possible french litterature, or to sum it up, or to establish what litterature should be or not. Instead, it will be an authority as to what are the rules of the french language and litterature, and keep a standard of it, not *inventing* it but rather making a synthesis of what exists, so that people speaking french do have a common reference.
viii. Similarly, the OS authorities should not provide the ONE TRUE PROGRAMS that will perform each single task in the system, but instead will maintain a public, open reference of how people are meant to communicate, and should be required to communicate whenever a disagreement appears that cannot be otherwise fairly settled.
ix. There need not be a single administration that would manage all laws and regulations. Instead, it is much better that various specialized administrations made of proficient people each manage the fields where their members are proficient.
x.
xi. Choosing people in each of these specialized administration is not

harder than choosing people in a one centralized administration: specialization restricts the choice of potential nominees, and should also modify the weight of voters according to their expected knowledge about how to judge proficiency on the specialized subject.

xii. Of course, there need be a constitutional means to settle disagreements, and this means eventually there is a ultimate authority (because all ultrafilters on finite sets are principal); but central arbitration doesn't mean central management at all. A central arbitration would not take any initiative by itself, and would not rule anything, only judge between alternative when asked to. Refering to it should be an exceptional event; when a submitted case clearly matches a field for which an established authority already exists, the central authority would always follow the opinion of the competent authority, so that people wouldn't argue over and over when the competent authority decided something.

xiii. If privacy was one of the constitutional principles, then laws can't uselessly constraining the private behavior of objects.

xiv. the protection-handling or proof-checking ”microkernel” is like the executive – it enforces the respect of the rules of the system.

Seeing how existing human States and computer kernels fail to do their job is left as an exercise to the reader.

The point is that all this infrastructure is meant to help objects (people) communicate with each other in fair terms, so that the global communication is faster, safer, and more accurate, with less noise, while consuming less resources. It should make the objects nearer to each other.

The role of State id to allow people to communicate.

To stay politically as neutral as possible (after all, this is a technical paperr), the paper should try to not explicitly use a reference to State, if possible. Instead, it would conclude with a note according to which the very same argument would hold when applied to human societies as similar dynamical systems.

(g) Contrarily to the socialists, who say that a state-ruled society is the End of History, the Authentic Liberals do not say that a free, fair, market is the end of history; on the contrary, they say that a free, fair, market, is the beginning of history; it is a prerequisite for information to pass well, for behaviors to adapt, for changes to operate, for history to exist. The freer, fairer the market, the more history.

(h) It may be said that computing has been doing quantitative leaps, but has not done any comparable qualitative leap; computing grows in extension, but does not evolve toward intelligence; it sometimes rather becomes more largely stupid. This is the problem of operating systems not having a good conceptual kernel: however large and complete their standard library, their utility will be essentially restricted to the direct use of the library.

A.10 Newest Operating Systems: the so-called ”Multimedia revolution”

This phenomenon can also be explained by the fact that programmers, long used to software habits from the heroic times when computer memories were too tight to contain more than just the specific software you needed (when they even could), do not seem to know how to fill today computers' memory, but with pictures of gorgeous women and digitized music (which *is* the so-called multimedia revolution). Computer hardware capabilities evolved much quicker than human software capabilities; thus humans find it simpler to fill computers with raw data (or almost raw data) than with intelligence.

Those habits, it must be said, were especially encouraged by the way information could not spread and augment the common *public* background, since because of lack of theory and practice of what a freely communicating world could or should be, only big companies enforcing "proprietary" label could up to now broadcast their software; people who would develop original software thus had (and sadly still have) to rewrite everything from almost scratch, unless they could afford a very high price for every piece of software they may want to build upon, without having much control on the contents of such software.

(i) The role of the OS infrastructure in the computer world is much like that of the State in human societies: it should provide justice by guaranteeing, by force if needs be, that contracts will be fulfilled, and nothing more. In the case of computer software, this means that it will guarantee that contracts passed between objects will be fulfilled, that objects should fulfill each other's requirements before they can connect. When there is no Justice, there is no society/OS, but only chaos.

(j) Because it ain't in the Kernel doesn't mean it ain't done. [Because the government doesn't do it doesn't mean nobody does it].

(k) The Kernel is there as a warrant that voluntarily agreed contracts between objects be respected: if function F is ready to trade a golden coin from some quantity of gold powder, the kernel will see that people trading with F will actually provide the right amount of gold powder, whereas F will actually return a gold coin.

A.11 Part II

(a) Part II would discuss programming language utility, stating the key concepts about it.

   i. Any reuse includes some rewrite, which is to minimize. Similarly, when we "rewrite", we often reuse a lot of the formal and informal ideas from existing code, and even when we reinvent, we reuse the inspiration, or sometimes feedback from people already inspired.

   ii. Notably after discussing how to be able to construct as many new concepts as possible, it should explain that the key to concept expressivity (that reflectivity cannot indefinitely postpone) is their separation power, and thus the capability to affirm one of multiple alternatives, to express different things, to negate and deny things.

(b) Literate Programming, and D.E. Knuth's attempts with WEB and C/WEB (see this interview of D.E. Knuth `http://www.clbooks.com/nbb/knuth.html`) are actually ways to pass more information about programs. To pass information that programming languages themselves don't/can't/can't-efficiently pass, through well-organized human-readable documentation. This is *A GOOD THING*, because there will *ALWAYS* be things that humans can (already) express that machines cannot express (yet). But this is *NOT THE PANACEA*, because there *ARE* things that the machines *ACTUALLY COULD* express with high-level languages, that pure literate programming over low-level languages require the human to not only to write, but to check, when a computer is much better suited to check them. Knuth completely ignores the meta- capabilities of computers.

(c) Each independent part is subject to the limit of a one-man's understanding so it be reliable. Existing systems are coarse-grained, which means that independent parts or large portions of programs, so that a complete program is made of few independent parts, and that total program complexity is limited to the sum of a few direct human understandings.

(d) Tunes will be fine-grained and reflective, so that a complete program is made of arbitrarily many limited parts, and can arbitrarily grow in complexity.

(e) Paradoxically, while their user-interface abstractions are coarse-grained and "high-level" (in a complex), current OSes only provide a very low-level set of programming abstractions to combine at a fine-grained level for any reliability/efficiency. There is double-speach here, and both users and programmers are hindered.

(f) If safety criteria are not expressible by the computer system, then to be safe, programs must be understandable by men. And because it is not essentially harder to express the criteria to the machine than to another man, this most likely means that a one man. Because that man won't ever be there to maintain code, because armies of "maintainers" won't replace him, because there is no tool to safely adapt old programs to new points of views, then every so often, code must go to the dust bin. No wonder why software evolves so slowly: only some small human experience remains, and even then, because there is no way to express what that experience is, it cannot spread in technology fast ways, but only man to man.

(g) Having generic programs instead of just specific ones is exactly the main point that we saw about having a good grammar to introduce new generic objects, instead of just an increasing number of terminal, first order objects, that actually do specific things (i.e. extending the vocabulary).

(h) What is really useful is a higher-order grammar, that allows to manipulate any kind of abstraction that does any kind of things at any level. We call level 0 the lowest kind of computer abstraction (e.g. bits, bytes, system words, or to idealize, natural integers). Level one is abstractions of these objects (i.e. functions manipulating them). More generally, level n+1 is made of abstractions of level n objects. We see that every level is a useful abstraction as it allows to manipulate objects that would not be possible to manipulate otherwise.

But why stop there ? Everytime we have a set of level, we can define a new level by having objects that arbitrarily manipulate any lower object (that's ordinals); so we have objects that manipulate arbitrary objects of finite level, etc. There is an unbounded infinity of abstraction levels. To have the full power of abstraction, we must allow the use of any such level; but why not allow manipulating such full-powered systems ? Any logical limit you put on the system may be reached one day, and this day, the system would become completely obsolete;

that's why any system to last must *potentially* contain (not in a subsystem) any single feature that may be needed one day.

The solution is not to offer any bounded level of abstraction, but unlimited abstracting mechanisms; instead of offering only terminal operators (BASIC), or first level operators (C), or even finite-order offer combinators of arbitrary order.

offer a grammar with an embedding of itself as an object. Of course, a simple logical theorem says that there is no consistent *internal* way of saying that the manipulated object is indeed the system itself, and the system state will always be much more complicated than it allows the system to understand about itself; but the system implementation may be such that the manipulated object indeed is the system. This is having a deep model of the system inside itself; and this is quite useful and powerful. This is what I call a higher-order grammar – a grammar defining a language able to talk about something it believes be itself. And this way only can full genericity be achieved: allowing absolutely anything that can be done about the system, from inside, or from outside (after abstracting the system itself).

..... First, we see that the same algorithm can apply to arbitrarily complex data structures; but a piece of code can only handle a finitely complex data structure; thus to write code with full genericity, we need use code as parameters, that is, *second order.* In a low-level language (like ”C”), this is done using function pointers.

We soon see problems that arise from this method, and solutions for them. The first one is that whenever we use some structure, we have to explicitly give functions together with it to explain the various generic algorithm how to handle it. Worse even, a function that doesn't need some access method about an the structure may be asked to call other algorithms which will turn to need know this access method; and which exact method it needs may not be known in advance (because what algorithm will eventually be called is not known, for instance, in an interactive program). That's why explicitly passing the methods as parameters is slow, ugly, inefficient; moreover, that's code propagation (you propagate the list of methods associated to the structure – if the list changes, all the using code changes). Thus, you mustn't pass *explicitly* those methods as parameters. You must pass them implicitly; when using a structure, the actual data and the methods to use it are embedded together. Such a structure including the data and methods to use it is commonly called an *object*; the constant data part and the methods, constitute the *prototype* of the object; objects are commonly grouped into *classes* made of objects with common prototype and sharing common data. *This* is the fundamental technique of *Object-Oriented* programming; Well, some call it that Abstract Data Types (ADTs) and say it's only part of the ”OO” paradigm, while others don't see anything more in ”OO”. But that's only a question of dictionary convention. In this paper, I'll call it only ADT, while ”OO” will also include more things. But know that words are not settled and that other authors may give the same names to different ideas and vice versa.

BTW, the same code-propagation argument explains why side-effects are an especially useful thing as opposed to strictly functional programs (see pure ML :); of course side effects complicate very much the semantics of programming, to a point that ill use of side-effects can make a program impossible to understand or debug – that's what not to do, and such possibility is the price to pay to prevent code propagation. Sharing *mutable* data (data subject to side effects) between different embeddings (different *users*) for instance is something whose semantics still have to be clearly settled (see below about object sharing).

The second problem with second order is that if we are to provide functions other functions as parameter, we should have tools to produce such functions. Methods can be created dynamically as well as ”mere” data, which is all the more frequent as a program needs user interaction. Thus, we need a way to have functions not only as parameters, but also as result of other functions. This is *Higher order*, and a language which can achieve this has a *reflective* semantics. Lisp and ML are such languages; FORTH also, whereas standard FORTH memory management isn't conceived for a largely dynamic use of such feature in a persistent environment. From "C" and such low-level languages that don't allow a direct portable implementation of the higher-order paradygm through the common function pointers (because low-level code generation is not available as in FORTH), the only way to achieve higher-order is to build an interpreter of a higher-order language such as LISP or ML (usually much more restricted languages are actually interpreted, because programmers don't have time to elaborate their own user customization language, whereas users don't want to learn a new complicated language for each different application and there is currently no standard user-friendly small-scale higher-order language that everyone can adopt – there are just plenty of them, either very imperfect or too heavy to include in every single application).

With respect to typing, Higher-Order means the target universe of the language is reflective – it can talk about itself.

With respect to Objective terminology, Higher-Order consists in having classes as objects, in turn being groupable in *meta-classes*. And we then see that it *does* prevent code duplication, even in cases where the code concerns just one user as the user may want to consider concurrently two – or more – different instanciations of a same class (i.e. two *sub-users* may need toe have distinct but mostly similar object classes). Higher-Order is somehow allowing to be more than one computing environment: each function has its own independant environment, which can in turn contain functions.

To end with genericity, here is some material to feed your thoughts about the need of system-builtin genericity: let's consider multiplexing. For instance, Unix (or worse, DOS) User/shell-level programs are ADTs, but with only one exported operation, the "C" main() function per executable file. As such "OS" are huge-grained, with ultra-heavy inter-executable-file (even inter-same-executable-file-processes) communication semantics no one can afford one executable per actual operation exported. Thus you'll group operations into single executables whose main() function will multiplex those functionalities.

Also, communication channels are heavy to open, use, and maintain, so you must explicitly pass all kind of different data & code into single channels by manually multiplexing them (the same for having heavy multiple files or a manually multiplexed huge file).

But the system cannot provide builtin multiplexing code for each single program that will need it. It does provide code for multiplexing the hardware, memory, disks, serial, parallel and network lines, screen, sound. POSIX requirements grow with things a compliant system oughta multiplex; new multiplexing programs ever appear. So the system grows, while it will never be enough for user demands as long as *all* possible multiplexing won't have been programmed, and meanwhile applications will spend most of their time manually multiplexing and demultiplexing objects not yet supported by the system.

Thus, any software development on common OSes is hugeware. Huge in hardware resource needed (=memory - RAM or HD, CPU power, time, etc), huge in resource spent, and what is the most important, huge in programming time.

The problem is current OSes provide no genericity of services. Thus they can never do the job for you. That why we really NEED *generic* system multiplexing, and more generally genericity as part of the system. If one generic multiplexer object was built, with two generic specializations for serial channels or flat arrays and some options for real-time behaviour and recovery strategy on failure, that would be enough for all the current multiplexing work done everywhere.

So this is for Full Genericity: Abstract Data Types and Higher Order. Now, if this allows code reuse without code replication – what we wanted – it also raises new communication problems: if you reuse objects especially objects designed far away in space or time (i.e. designed by other people or an other, former, self), you must ensure that the reuse is consistent, that an object can rely upon a used object's behaviour. This is most dramatic if the used object (e.g. part of a library) comes to change and a bug (that you could have been aware of – a quirk – and already have modified your program accordingly) is removed or added. How to ensure object combinations' consistency ?

Current common "OO" languages are not doing much consistency checks. At most, they include some more or less powerful kind of type checking (the most powerful ones being those of well-typed functional languages like CAML or SML), but you should know that even powerful, such type checking is not yet secure. For example you may well expect a more precise behavior from a comparison function on an ordered class 'a than just being `'a->'a->{LT,EQ,GT}` i.e. telling that when you compare two elements the result can be "lesser than", "equal", or "greater than": you may want the comparison function to be compatible with the fact of the class to be actually ordered, that is $x < y \& y < z => x < z$ and such. Of course, a typechecking scheme, which is more than useful in any case, is a deterministic decision system, and as such cannot completely check arbitrary logical properties as expressed above (see your nearest lectures in Logic or Computation Theory). That's why to add such enhanced security, you must add non-deterministic behaviour to your consistency checker or ask for human help. That's the price for 100% secure object combining (but not 100% secure programming, as human error is still possible in misexpressing the requirements for using an object, and the non-deterministic behovior can require human-forced admission of unproved consistency checks by the computer).

This kind of consistency security by logical formal property of code is called a formal specification method. The future of secure programming lies in there (try enquire in the industry about the cost of testing or debugging software that can endanger the company or even human lives if ill written, and insurance funds spent to cover eventual failures - you'll understand). Life concerned industries already use such modular formal specification techniques.

In any cases, we see that even when such methods are not used automatically by the computer system, the programmer has to use them manually, by including the specification in comments or understanding the code, so he does computer work.

Now that you've settled the skeleton of your language's requirements, you can think about peripheral deduced problems.

.....

(i) When the best fit technique is known, only this technique, and none else, should be used. any other use may be expedient, but not quite useful.

Moreover, it is very hard to anticipate one's future needs; whatever you do, there will always be new cases you won't have.

lastly, it doesn't replace combinators And finally, as of the combinatorials allowed allowing local server objects to be saved by the client is hard to implement eficiently without the server becoming useless, or creating a security hole;

..... At best, your centralized code will provide not only the primitives you need, but also the combinators necessary; but then, your centralized code is a computing environment by itself, so why need the original computing environment ? there is obviously a problem somewhere; if one of the two computing environment was good, the other wouldn't be needed !!!; All these are problems with servers as much as with libraries.

(j) With a long training, people can avoid most bugs that typing would have detected; but this long training has a human cost. And even then, all bugs are not *guaranteed* to be avoided, so insurance is still needed against huge occasional catastrophes, which also involves a high, non-linear cost.

Actually, the same holds for any kind of static information that might have been gathered about programs: you can live without the computer checking it, by checking it yourself. But then you must do computer work, are not guaranteed to do it properly, and cannot offer the guarantee to your customers, as youuur proof is all inside your mind and not repeatable!!!

(k) Paul R Wilson said:

BTW, this whole wrangle is exactly why I recommend avoiding the term "weakly typed." It means at least three different things to different people, and various combinations to other people:

i. dynamic typing
ii. implicit conversions, and
iii. unchecked types

(l) i. implicit vs explicit is what differentiates a HLL from a LLL. A LLL will require the pow
ii. not building an artificial border between programmers and users ⇒ not only the system programming *language* must be OO, but the whole *system*.
iii. easy user extensibility → language-level reflection.
iv. sharing mutable data: how ? → specifications & explicitly mutable/immutable (or more or less mutation-prone ?) & time & locking – transactions.
v. objects that *must* be shared: all the hardware resources – disks & al.
vi. sharing accross time → persistence - reaching precision/mem/speed/resource limit: what to do ? → exceptions
vii. recovering from exceptional situations: how ? → continuations (easy if higher-order on)
viii. tools to search into a library → must understand all kind of morphism in a logically specified structure.
ix. sharing accross network → distribution
x. almost the same: tools for merging code → that's tricky. Very important for networks or even data distributed on removable memory (aka floppies) – each object should have its own merging/recovery method.
xi. more generally tools for having side effects on the code.

(m) A.12 Structures

we consider Logical Structures: each structure contains some types, and symbols for typed constants, relations, and functions between

those types. Then we know some algebraic properties verified by those objects, i.e. a structure of typed objects, with a set of constants & functions & relations symbols, et al.
A structure A is interpreted in another structure B if you can map the symbols of A with combinations of symbols of B (with all the properties conserved). The simplest way to be interpreted is to be included.
A structure A is a specialization of a structure B if it has the same symbols, but you know more properties about the represented objects.

A.13 Mutable objects

We consider the structure of all the possible states for the object. The actual state is a specialization of the structure. The changing states accross time constitute a stream of states.

A.14 Sharing Data

The problem is: what to do if someone modifies an object that others see ? Well, it depends on the object. An object to be shared must have been programmed with special care.
The simplest case is when the object is atomic, and can be read or modified atomically. At one time, the state is well defined, and what this state is what other sharers see.
When the object is a rigid structure of atomic objects, well, we assume that you can lock parts of the object that must be changed together – in the meantime, the object is unaccessible or only readable – and when the modification is done, everyone can access the object as before. That's transactions.
Now, what to do when the object is a very long file (say text), that each user sees a small part of it (say a full screen of text), and that someone somewhere adds or deletes some records (say a sentence) ? Will each user's screen scroll according to the number of records deleted ? Or will they stay at the same spot ? The later behaviour seem more natural. Thus, a file has this behaviour that whenever a modification is done, all pointers to the file must change. But consider a file shared by *all* the users across a network. Now, a little modification by someone somewhere will affect everyone ! That's why both the semantics and implementation of shared objects should be thought about longly before they are settled.

A.15 Problem: recovery

What to do when assumptions are broken by higher priority objects ? e.g. when the user interrupts a real-time process, when he forces a modification in an otherwise locked file, when the process is out of memory, etc.
Imagine that a real-time process is interrupted for imperative reasons (e.g. a cable was unplugged; a higher-priority process took over the cpu, etc): will it continue where it stopped ? or will it skip what was done during the interruption ? Imagine the system runs out of memory ? Whose memory are you to reclaim back ? To the biggest process ? The smallest ? The oldest ? The lowest real-time priority ? The first to ask for more ? Or will you "panic" like most existing OSes ? If objects spawn, thus filling memory (or CPU), how to detect "the one" responsible and destroy it ?

If an object locks a common resource, and then is itself blocked by a failure or other unwilling latency, should this transaction be cancelled, so others can access the resource, or should all the system wait for that single transaction to end ?

As for implementation methods, you should always be aware that defining those abstraction as the abstractions they are, rather than hand-coded emulation for these, allows better optimizations by the compiler, quicker write phase for the programmer, neater semantics for the reader/reuser, no implementation code propagation for the reimplementer, etc.

Partial evaluation should also allow specialization of code that don't use all the language's powerful semantics, so that standalone code be produced without including the full range of heavy reflective tools.

(n) all the requirements to be used as for Tunes, or design a new one if none is found.

(o) That is, without ADTs, and combinating ADTs, you spend most of your time manually multiplexing. Without semantic reflection (higher order), you spend most of your time manually interpreting runtime generated code or manually compiling higher order code. Without logical specification, you spend most of your time manually verifying. Without language reflection, you spend most of your time building user interfaces. Without small grain, you spend most of your time manually inlining simple objects into complex ones, or worse, simulating them with complex ones. Without persistence, you spend most of your time writing disk I/O (or worse, net I/O) routines. Without transactions, you spend most of your time locking files. Without code generation from constraints, you spend most of your time writing redundant functions that could have been deduced from the constraints.

To conclude, there are essentially two things we fight: lack of feature and power from software, and artificial barriers that misdesign of former software build between computer objects and others, computer objects and human beings, and human beings and other human beings.

(p) **A.16 Centralized code**

There's been a craze lately about "client/server" architecture for computer hardware and software. What is "client/server" architecture that many corporations boast about providing ?

.....

conceptually, a server is a centralized implementation for a library; centralized $\Rightarrow$ coarse-grained; now, coarse grained $\Rightarrow$ evil; hence centralized $\Rightarrow$ evil. we also have centralized $\Rightarrow$ network bandwidth waste. only "advantage": the concept is simple to implement even by the dumbest programmer. Do corporations boast about their programmers being dumb ?

.....

A very common way to share code is to write a code "server" that will include tests for all the different cases you may need in the future and branch to the right one. Actually, this is only some particular kind of library making, but much more clumsy, as a single entry point will comprise all different behaviours needed. This method proves hard to design well, as you have to take into account all possible cases to arise, with predecided encoding, whereas a good encoding would have to take into account actual use (and thus be decided after run-time measurements). The obtained code is slow as it must test many uncommon cases; it is huge, as it must take into account many cases, most of them seldom or never actually used; it is also uneasy to use, as you must encode and decode the arguments to fit its one entry point's calling convention. It is very difficult to modify, but by adding new entries and replacing obsolete subfunctions by stubs, because it would else break existing code; it is very clumsy to grant partial access to the subfunctions, as you must filter all the calls; security semantics become very hard to define.

Centralized code is also called "client-server architecture"; the central code is called the server, while those who use it are called clients. And we saw that a function server is definitely something that no sensible man would use directly; human users tend to write a library that will encapsulate calls to the server. But it's how most operating systems and net-aware programs are implemented, as it's the simplest implementation way. Many companies boast about providing client-server based programs, but we see there's nothing to boast about it; client-server architecture is the simplest and dumbest mechanism ever conceived; even a newbie is able to do that easy. What they could boast about would be *not* using client-server architecture, but truely distributed yet dependable software.

A server is nothing more than a bogus implementation for a library, and shares all the disadvantages and limits of a library, with enhanced extensibility problem, and additional overhead. It's only advantage is to have a uniform calling convention, which can be useful in a system with centralized security, or to pass the stream of arguments through a network to allow distant client and servers to communicate. This last use is particularly important, as it's the simplest trick ever found for accessing an object's multiple services through a single communication line. Translating software interface from library to server is called multiplexing the stream of library/server access, while the reverse translation is called demultiplexing it.

A.17 Genericity

Then what are "intelligent" ways to produce reusable, easy to modify code? Such a method should allow reusing code without duplicating it, and without growing it in a both unefficient and uncomplete way: an algorithm should be written once and for once for all the possible applications it may have, not for a *specific* one. We have just found the answer to this problem: the opposite of specificity, *genericity.*

So we see that system designers are ill-advised when they provide such specific multiplexing, that may or may not be useful, whereas other kind of multiplexing is always needed (a proof of which being people always boasting about writing – with real pain – "client/server" "applications"). What they really should provide is *generic* ways to automatically multiplex lines, whenever such thing is needed.

More generally a useful operating system should provide a generic way to share resources; for that's what an operating system is all about: sharing disks, screens, keyboards, and various devices between multiple users and programs that may want to use those accross time. But genericity is not only for operating systems/sharing. Genericity is useful in any domain; for genericity *is* instant reuse: your code is generic – works in all cases – so you can use it in any circumstances where it may be needed, whereas specific code must be rewritten or readapted each new time it must be used. Specificity may be expedient; but only genericity is useful on the long run.

Let us recall that genericity is the property of writing things in their most generic forms, and having the system specialize them when needed, instead of hard-coding specific values (which is some kind of manual evaluation).

Now, *How* can genericity be achieved ?

(q) Machines can already communicate; but with existing "operating systems" the only working method they know is "client/server architecture", that is, everybody communicating his job to a one von Neuman machine to do all the computations, which is limited by the same technological barrier as before. The problem is current programming technology is based on coarse-grained "processes" that are much too heavy to communicate; thus each job must be done on a one computer.

(r) There is lots of laughable hype about network computers (NCs). NCs are the hardware embodiment of the client/server architecture: you plug NCs on the net, and the only configuration needed, that can be done automatically, is assigning them a network name/address. All the data is on the server side.

This just has all the advantages and disadvantages of the client/server: surely this ensures consistency of data, but efficiency is the worst possible among systems that ensure it, because of centralization.

An efficient system would achieve consistency of installed software without sacrificing performance, and without requiring a modification of current hardware, by doing all the consistency enforcement in software. Memory that is local to network hosts is then used for data cacheing and replication; CPU resources can be used to do distributed computations, etc. A software solution could do things great. A hardware solution like NCs is just waste of resources.

All the more, NCs do not even have compatibility, and do not allow any particular leverage of existing software, so that they are *really* a big gratuitous waste of resources. For a fraction of the price of all the wasted hardware resources, a proven distributed OS could be developed and marketed!

(s) i. features: high-level abstraction Real-time, reflective language frame, code & data persistence, distribution, higher order

ii. misfeatures: low-level abstraction explicit batch processing, adhoc languages, sessions & files, networking, first order

(t) because many semantical changes are to be manually propagated accross the whole program.

(u) "Compilers can not guarantee a program won't crash." have I been told.

Surely, given an expressive enough language, there is no compiler that can tell for an arbitrary program whether it will crash or not.

Happily, computers are not used to run random arbitrary programs!!! Well, there's genetic programming and corewar, fair enough. When you program,

i. you know what you want, so you know what a correct program is, and more easily even what a non-crashing program is.

ii. you write your program specifically so you can prove to yourself that the program is correct.

iii. if programming under contract, you are expected to have done the former, even though the customer has no way to check, but perhaps having you fill red tape.

Well, instead of all these knowledge and proofs to stay forever untold and unchecked, it is possible to use a language that can express them all! A computer language could very well express all the requirements for the program, and logically check a proof that they are fulfilled.

A.18 Part III

(a) Part III would apply the concepts to existing technology.

   i. It would have to discuss what is tradition, what is its role, how it should or not be considered, what it currently does wrong, how the Tunes approach inserts in it.
   ii. It would debunk myths
   iii. Efficiency,Security,Small-grain: take two, you have the third. That is, when you need two of them, you also need the third, but when you have two of them, you automatically have the third.
   iv. with OO, people discovered that implicit binding is needed. Unhappily, most "OO" only know as-late-as-possible binding and no such thing as reflectivity (=implicitness control) or migration (=modification of implicitness control).

(b) Name:

   i. "Tradition and Revolution" ?
   ii. "Hierarchy vs Liberty" ?
   iii. "Myths and reality" ?
   iv. "The burden of the past" ?
   v. "No computer is an island, entire in itself" ?

(c) Draft:

   i. This part would explain how we apply the principles from part I and II to actual computing
   ii. It would recall what tradition is, what the two meanings for revolution are, and why a one applies and not the other.
   iii. It would try debunk some myths:
   iv. Tapes vs Files vs Persistency
   v. Linear ASCII Text vs hypertext vs Meta-text,
   vi. Single-Computer OS vs Networked OS vs Distributed OS
   vii. Single-User vs Multi-user vs dynamic user
   viii. console vs GUI vs decoupling of programming and IO
   ix. They are all instances of the "Flat Resource vs Hierarchical Layering vs Higher-order modularity" paradigm.

(d) Text as source files: derives from the silly notion that because people type programs by hitting a sequence of keys, each being marked with a symbol, the program should be stored and manipulated as the sequence of symbol typed. That because books are read as such a sequence, because time is linear and thus any exploration of the text, then the text should be linear, too. This also derives from the fact that early computers where so slow and primitive, with such tight memory, that programmers had to care a lot about the representation of computer data, with the sequential nature of their being fed to the computer through punched paper or magnetic tape. Derives from belief that the object is its representation, and that the only valid representation is the "usual" one.

(e) The Web allows casual users to publish new information from where they are. This is quite a progress. But only passive documents can be published; any inter-site reference is unreliable. The most advanced programming techniques (cgi-bin) only allow unsafe localized low-level computations.

(f) A common myth about programming is that low-level programming allows more efficiency than high-level programming. This is completely untrue, while the opposite is quite true. Actually, people spend several million dollars at developping optimizing C and FORTRAN compilers, but a much cheaper Common LISP compiler (CMU Common LISP, developped by a few students and teachers), achieve similar performance, while allowing the whole expressivity of a real high-level language. Also, people may see that a large part of modern optimizers consist in making the whole code higher-level, so it can be better understood and optimized by the compiler. Any amount of time spent at manually optimizing some routine, could be equally spent at developping some specialized optimizing heuristics of same effect on the particular low-level routine, but that could generalize to further modified versions of the routine, or of similar routines, thus improving reliability and maintainability as well as performance, and saving a lot of time. Of course, this means that compiler technology with the ability to accept user-defined optimizing heuristics be widely available. But this is just possible and will be case. Instead of losing ever more time at low-level coding, most low-level people should consider making such a compiler appear sooner. Actually, a trivial theoretical argument could have told us that already: high-level programs contain more information and less noise than low-level programs, hence, can be manipulated and compiled more efficiently, with proper tools; and anything that can be done in low-level can be done at least as well, and surely more cleanly and genericly, in high-level.

(g) Axioms:

   i. "No man should do what the computer can do quicker for him (including time spent to have the computer understand what to do)" – that's why we need to be able to give order to the computer, i.e. to program.

   ii. "Do not redo what others already did when you've got more important work" – that's why we need code reuse.

   iii. "no uncontrolled code propagation" – that's why we need genericity.

   iv. "security is a must when large systems are being designed" – that's why we need strong typechecking and more.

   v. "no artificial border between programming and using" – that's why the entire system should be OO with a unified language system, not just a hidden system layer.

   vi. "no computer user is an island, entire by itself" – you'll always have to connect (through cables, floppies or CD-ROMs or whatever) to external networks, so the system must be open to external modifications, updates and such.

(h) Current computers are all based on the von Neumann model in which a centralized unit executes step by step a large program composed of elementary operations. While this model is simple and led to the wonderful computer technology we have, laws of physics limit in power future computer technology to no more than a grand maximum factor 10000 of what is possible today on superdupercomputers.

   This may seem a lot, and it is, which leaves room for many improvement in computer technology; however, the problems computer are confronted to are not limited anyway by the laws of physics. To break this barrier, we must use another computer model, we must have many different machines that cooperate, like cells in a body, ants in a colony, neurones in a brain, people in a society.

(i) More than 95about Interfaces: interfaces with the system, interfaces with the human. Actual algorithms are very few, heuristics are at the same time few and too many, because the environment makes them unreliable. Interfaces can and should be semi-automatically deduced.

(j) More generally, the problem with existing systems is lack of reflectivity, and lack of consistency: you can't simply, quickly, reliably, automate any kind of programming. in a way such that system consistency be enforced.

(k) Persistence is necessary for AI:

   i. Intelligence is the fruit of a long tradition. Even a most intelligent and precocious human being must be carefully bred for years before yielding the faintest result.
   ii. How could you expect a machine to become intelligent as soon as it is built and powered-up, or even after being powered-up for some hours, or some days ?
   iii. computers currently do not allow any information to persist reliably more than a few months, and won't translate information from old software to newer ones.
   iv. Hence, artificial intelligence is not possible with existing architecture.
   v. However, systems with persistent memory could be a first step toward AI.

(l) unindustrialized countries: the low reliability of power feeds make resiliant persistency a must.

(m) Why are existing OS so bad ? For the same reason that ancient lore is completely irrelevant in nowadays' world:

   i. At a time when life was hard, memories very small and expensive, development cost very high, people had to invent hacker's techniques to survive; they made arbitrary decisions so survive with their few resources; They behaved dirtily, and thought for the short term.
   ii. They just had to.
   iii. Now, technology has always evolved at an increasing pace. What was experimental truth is always becoming obsolete, and good old recipes are becoming out of date. Behaving cleanly and thinking for the long term is made possible.
   iv. It is made compulsory.
   v. The problem is, most people don't think, but blindly follow traditions. They do not try to distinguish what is truth and what is falsehood in traditions, what is still true, and what no longer stands. They take it as a whole, and adore it religiously, sometimes by devotion, most commonly by lack of thinking, often by refusal to think, rarely but already too often by a hypocrit calculus. Thus, they abdicate all their critical faculties, or use it against any ethics. As a result, for the large majority of honest people, their morals are an unspeakable burden, mixing common sense, valid or obsolete experimental data, and valid, outdated, or false rules, connected and tangled in such a way that by trying to extract something valid, you come up with a mass of entangled false things that are associated, and that when extirping false things, you often destroy the few that were valid together. The roots of their opinions are not in actual facts, but in lore, hence their being only remotely relevant to anything.
   vi. Tunes intends to rip off all these computer superstitions.

(n)
A.19 Down to actual OSes

.....

A.20 Humanly characteristics of computers

persistence, resilience, mobility, etc....
response to human

(o) The Internet is a progress, in that people can publish documents. But these documents are mostly passive. Those that are not suppose highly-qualified specialists to care about;

A.21 Multiplexing: the main *runtime* activity of an OS

Putting aside our main goal, that is, to see how reuse is possible in general, let us focus on this particular multiplexing technique, and see what lessons we can learn that we may generalize later.

Multiplexing means to split a single communication line or some other resource into multiple sub-lines or subresources, so that this resource can be shared between multiple uses. Demultiplexing is recreating a single line (or resources) from those multiple ones; but as dataflow is often bi-directional, this reverse step is most often unseparable from the first, and we'll only talk about multiplexing for these two things. Thus, multiplexing can be used to share a multiple functions with a single stream of calls, or convertly to have a function server be accessed by multiple clients.

Traditional computing systems often allow multiplexing of *some* physical resources, thus spliting them into a first (but potentially very large) level of equivalent logical resources. For example, a disk may be shared with a filesystem; CPU time can be shared by task-switching; a network interface is shared with a packet-transmission protocol. Actually, what any operating system does can be considered multiplexing. But those same traditional computing systems do not provide the same multiplexing capability for arbitrary resource, and the user will eventually end-up with having to multiplex something himself (see the term user-level program to multiplex a serial line; or the screen program to share a terminal; or window systems, etc), and as the system does not support anything about it, he won't do it the best way, and not in synergy with other efforts.

What is wrong with those traditional systems is precisely that they only allow limited, predefined, multiplexing of *physical* resources into a small, predefined, number of *logical resources*; there they create a big difference between physical resources (that may be multiplexed), and logical ones (which cannot be multiplexed again by the system). This gap is completely arbitrary (programmed computer abstractions are never purely physical, neither are they ever purely logical); and user-implemented multiplexers must cope with the system's lacks and deficiencies.

(p) More generally, in any system, for a specialized task, you may prefer dumb workers that know well their job to intelligent workers that that cost a lot more, and are not so specialized. But as the tasks you need to complete evolve, and your dumb workers don't, you'll have to throw them away or pay them to do nothing as the task they knows so well is obsolete; they may look cheap, but they can't adapt, and their overall cost is high for the little time when they are active;

In a highly dynamic world, you lose at betting on dumbness, and should invest on intelligence.

whereas with the intelligent worker, you may have to invest in his formation, but will always have a proficient collaborator after a short adaptation period. After all, even the dumb worker had to learn one day, and an operating system was needed as a design platform for any program.

(q) People tend to think statically in many ways.

(r) At the time when the only metaprogramming tool was the human minds of specialized engineers, because memories were too small, which is very expensive and cannot deal with too much stuff at once, a run-time hardware protection was wishable to prevent bugs in existing programs from destroying data, even though th But now that computers have enough horsepower to be useful metaprogrammers, the constraints change completely.

(s) Dispell the myth of ”language independence”, particularly about OSes. which really means ”interfaces to many language implementations”; any expressive-enough language can express anything you expressed in another language in many ways.

(t) And as the RISC/CISC then MISC/RISC concepts showed, the best way to achieve this is to keep the low-level things as small as possible, so as to focus on efficiency, and provide simple (yet powerful enough) semantics. The burden of combining those low-level things into useful high-level objects is then moved to compilers, that can do things much better than humans, and take advantage of the simpler low-level design.

(u) Now, the description could be restated as: ”project to replace existing Operating Systems, Languages, and User Interfaces by a completely rethough Computing System, based on a correctness-proof-secure higher-order reflective self-extensible fine-grained distributed persistent fault-tolerant version-aware decentralized (no-kernel) object system.”

(v) GC&Type checking need be in developing version, not forcibly in developed version.

(w) Nobody should be *forced* by the system itself into proving one’s program correctness with respect to any specification. Instead, everyone is *enabled* to write proofs, and can *require* proofs from others.

Thus, you can know precisely what you have and what you don’t when you run code. When the code is safe, you know you can trust it. When it ain’t, you know you shouldn’t trust it.

Surely, you will object that because of this system, such man will now require you to give a proof that you can’t or won’t give to him, so NOW you can’t deal with him anymore. But don’t blame it on the system. If the man wants the proof, it means he’d expected your provided software to behave accordingly in the past, but just couldn’t require a proof, which was impossible. By dealing with the man, you’d have been morally and/or legally bound to provide the things that he now asks a proof for. Hence the proofable system didn’t deprive you from making any lawful thing. It just helped formalize what is lawful and what isn’t.

If the man requires so difficult proofs that he can’t find any provider to that, he will have to adapt, die, or pay more.

If the man’s requirements are outrageously excessive, and no-one should morally provide him the proofs, then he obviously is a nasty fascist pig, or whatever, and it’s an improvement that no-one will now deal with him.

To sum up things, being able to write/read/provide/require proofs means being able to transmit/receive more information. This means that people can better adapt to each other, and any deal that the system will cancel was an unlawful deal, replaced by better deals. Hence this technology increases the expressivity of languages, and does not decrease it. The system won’t have any statical specification, but will be a *free market* for people having specifications and people having matching software to safely exchange code against money, instead of being a *blind racket*.

(x) People like that the cryptic Perl syntax be ambiguous and guess what you mean from context, because it allows rapid development of small programs, and Perl usually guesses right what you want it to do.

other people will object that because your programs will then depend on guesses, you can’t reliably develop large programs and be confident that you don't depend on a guess that may prove wrong in certain conditions, or after you modify your program a bit.

But why should you *depend* on dynamic guesses? A Good programming language would allow you

i. to control how the guesses are done, enable some tactics, disable some others, and write your own.
ii. to make the guesses explicitly appear or disappear in the program by automatic semantic-preserving source-to-source transformations.
iii. resolve all the ambiguities in a static way through some interactive tool, with a reasonable guess as the default, but with the program's source being statically disambiguated by the machine.

Of course, all this require a much more reflective platform than we have, with interactive tools being much more integrated to the compiler than currently is.

(y) an open system, where computational information can efficiently flow with as little noise as possible.

Open system means that people can contribute any kind of information they want to the available cultural background, without having to throw everything away and begin from scratch, because the kind of information they want to contribute does not fit the system.

Example: I can't have lexical scopes in some wordprocessor spell-checker, only one "personalized dictionary" personalized at once (and even then, I had to hack a lot to have more than one dictionary, by swapping a unique global dictionary). So bad. I'll have to wait for next version of the software. Because so few ask for my feature, it'll be twenty years until it makes it to an official release. Just be patient. Or if I've got lots of time/money, I can rewrite the whole wordprocessor package to suit my needs. Wow!

On an open system, all software components must come in small grain, with possibility of incremental change anywhere, so that you can change the dictionary-lookup code to handle multiple dictionaries merged by scope, instead of a unique global one, without having to rewrite everything.

Current attempts to build an open system have not been fully successful. The only successful approach to offer fine-grained control on objects has been to let sources freely available, allowing independent hackers/developers to modify and recompile; but apart from the object grain problem, this doesn't solve the problems of open software. Other problems include the fact This offers no semantical control of seamless data conservation accross code modification; contributions are not really incremental in that the whole software must be integrally recompiled, stopped, relaunched; Changes that involve propagation of code among the whole program cannot be done incrementally with non because they

(z) "as little noise as possible": this means that algorithmic information can be passed without any syntactical or architectural constraint in it that would not be specifically intended; that people are never forced to say either more than they mean or less than they mean.

Example: with low-level languages like C, you can't define a generic function to work on any integer, then instanciate to the integer implementation that fits the further problem. If you define a function to work on some limited number type, then it won't work on longer numbers than the limit allows, while being wasteful when cheaper more limited types might have been used. Then if some 100000 lines after, you see that after all, you needed longer numbers, you must rewrite everything, while still using the previous version for existing code. Then you'll have two versions to co-debug and maintain, unless you let them diverge inconsistently, which you'll have to document. So bad. This is being required to say too much.

And of course, once the library is written, in a way generic enough so it can handle the biggest numbers you'll need (perhaps dynamically sized numbers), then it can't take advantage of any particular situation where the known constraints on numbers could save order of magnitudes in computations; of course, you could still rewrite yet another version of the library, adapted to that particular knowledge, but then you again have the same maintenance problems as above. This is being required to say too little.

Any "information" that you are required to give the system before you know it, without your possibly knowing it, without your caring about it, with your not being able to adjust it when you further know more, all that is *noise*.

Any information that you can't give the system, because it won't heed it, refuse it as illegal, implement in so inefficient a way that it's not usable, is *lack of expressiveness*.

Current languages are all very noisy and inexpressive. Well, some are even more than others.

The "best" available way to circumvent lack of expressiveness from available language is known as "literate programming", as developed, for example, by D.E.Knuth with his WEB and C/WEB packages. With those, you must still fully cope with the noise of a language like C, but can circumvent its lack of expressiveness, by documenting in informall human language what C can't express about the intended use for your objects.

Only there is no way accurately verify that objects are actually used consistently with the unformal documented requirements, which greatly limits the (nonetheless big) interest of such techniques; surely you can ask humans to check the program for validity with respect to informal documentation, but his not finding a bug could be evidence for his unability to find a real bug, as well as the possible absence of bug, or the inconsistency of the informal documentation. This can't be trusted remotely as reliably as a formal proof.

The Ariane V spacecraft software had been human-checked thousands of times against informal documentation, but still, a software error would have \$ 10^9 disappear in fumes; from the spacecraft failure report, it can be concluded that the bug (due to the predictable overflow of an inappropriately undersized number variable) could have been *trivially* pin-pointed by formal methods! Please don't tell me that formal methods are more expensive/difficult to put in place than that the rubbish military-style red-tape-checking that was used in place.

As a french taxpayer, I'm asking immediate relegation of the responsible egg-heads to a life-long toilet-washing job (their status of french "civil servants" prevents their being fired). Of course my voice is unheard.

Of course, there are lots of other software catastrophes that more expressive languages would have avoided, but even this single 10 G\$ crash would pay more than it would ever cost to develop formal methods and (re)write all critical software with!

(a) It is amazing that researchers in Computer Science are not developing branches of a same software, but everytime rewriting it all from scratch. For instance, people experimenting with persistence, migration, partial evaluation, replication, distribution, parallelization, etc, just cannot write their part independently from the others. Their pieces of code cannot combine, and if each isolated technical point is proven possible, they are never combined, and techniques can never really be fairly compared, because they are never applicable to the same data.

It is as if mathematicians would have to learn a completely new language, a completely new formalism, a completely new notation, for every mathematical theory! As if every book couldn't actually assume results from other books, unless they were proven again from scratch.

It is as if manufacturers could not assemble parts unless they all came from the same factory.

Well, such phenomena happen in other place than computer software, too. Basically, it might be conceived as a question of lack of standards. But it's much worse with computer software, because computer software is pure information. When software is buggy, or unable to communicate, it's not worth a damn; it ain't even good as firewood, as metal to melt or anything to recycle.

Somehow, the physical world is a universal standard for the use of physical objects. There's no such thing in the computer world where all standards are conventional.

Worse, progress in hardware and software implementation techniques is incompatible with the advent of definitive computerware standards, so that either standards are made ephemeral, or implementational progress is throttled.

And the solution is Reflection.

(b) The other day, I tried to explain what Reflection is to a mathematician friend of mine. But Reflection is so natural a thing for mathematicians (and my math background is perhaps what makes it hard for me to live without it in the computer world), that I could only try to describe what lack of Reflection would be in math:

It would mean that you could only combine theorems that were not developped with the very same formalism. For instance, you would not even be able to apply to classic results, unless you could provide some actual derivation of both well-known theorems from scratch using the same formalism, which for a mathematician would mean a conventional minimalistic theory like peano's axioms for arithmetics, or some flavor of set theory.

This very notion of "scratch" will seem silly to a mathematician, as he knows from Goedel that there is no absolute "scratch" from which to build mathematics, so that theorems should instead be produced in whatever form seems the most adequate for its current use (its being proved or reused, etc).

The computer engineer would then say that "scratch" is the actual operating software/hardware he's got to work on NOW, which is very concrete. But this notion of scratch should be silly to him, too, if only he were conscious how fast hardware technology evolves, and building from current "scratch" only ties his programs to current technology, preventing computerware upgrade, or limiting the benefits thereof.

Reflection does allow to refine implementations, to move between standards, to prove new meta-theorems and use them, to juggle between representations so as to pick up the most adequate for a given task without sacrificing consistency, to reuse (meta)theorems from other people, etc.

A.22 Miscellaneous notes

i. I saw your answer about an article in the news, so i wanna know, what is Tunes ?
   Well, that's a tough one. Here is what I told Yahoo: "TUNES is a project to replace existing Operating Systems, Languages, and User Interfaces by a completely rethought Computing Sys-

tem, based on a correctness-proof-secure higher-order reflective self-extensible fine-grained distributed persistent fault-tolerant version-aware decentralized (no-kernel) object system."

Now, there are lots of technical terms in that. Basically, TUNES is a project that strives to develop a system where computists would be much freer than they currently are: in existing systems, you must suffer the inefficiencies of

A. centralized execution [=overhead in context switching],
B. centralized management [=overhead and single-mindedness in decisions],
C. manual consistency control [=slow operation, limitation in complexity],
D. manual error-recovery [=low security],
E. manual saving and restoration of data [=overhead, loss of data],
F. explicit network access [slow, bulky, limited, unfriendly, unefficient, wasteful distribution of resource],
G. coarse-grained modularity [=lack of features, difficulty to upgrade]
H. unextensibility [=impossibility to do things oneself, people being taken hostage by software providers]
I. unreflectivity [=impossibility to write programs clean for both human and computer; no way to specify security]
J. low-level programming [=necessity to redo things again everytime one parameter changes].

If any of these seems unclear to you, I'll try to make it clearer in

ii. Note that Tunes does not have any particular technical aim *per se*: any particular technique intended for inclusion in the system has most certainly already been implemented or proposed by someone else already, even if we can't say where or when. Tunes does not claim any kind of technical originality. Tunes people are far from being the most proficient in any of the technical matters that they'll have to use, and hope that their code will be eventually replaced by better code written by the best specialists wherever applicable. But Tunes is not an empty project for that. Tunes does claim to bring some kind of original information, just not of a purely technical nature, but instead, as a global frame to usefully combine those various techniques as well as arbitrary future ones into a coherent system, rather than have them stay idle gadgets that can't reliably communicate with each other. We Tunes people hope that our real contribution will be the very frame in which the code from those specialists can positively combine with each other, instead of being isolated and helpless technical achievements. Even if our frame doesn't make it into a worldwide standard, we do hope that our effort will make such a standard appear sooner than it would have without us (if it ever would), and avoid the traps that we'll have uncovered.

iii. In this article, we have started from a general point of view of moral Utility, and by applying it to the particular field of computing, we have deduced several key requirements for computing systems to be as useful as they could be. We came to affirm concepts like dynamism, genericity, reflectivity,

separation and persistency, which unhappily no available computing system fully implements.
So to conclude, there is essentially one thing that we have to fight: the artificial informational barriers that lack of expressivity and misdesign of former software, due misknowledge, misunderstanding, and reject of the goals of computing, build between computer objects and others computer objects, computer objects and human beings, human beings and other human beings.

iv. People are already enough efficiency-oriented so that TUNES needn't invest a lot in it, just providing a general frame for others to insert optimization into. In the case of a fine-grained dynamic reflective system, this means that hooks for dynamic partial evaluation must be provided. This is also an original idea, that hasn't been fully developed. contribution that TUNES

v. When confronted with some proposition in TUNES, people tend to consider it separated from the rest of the TUNES ideas, and they then conclude that the idea is silly, because it contradicts something else in the traditional system design. These systems indeed have some coherency, which is why they survived and were passed by tradition. But TUNES tries to be much more coherent even,

vi. When I begun this article, long ago, I believed that multiplexing was the main thing an OS would do. Now, I understand that the main thing is trust. Multiplexing can be readily done with a powerful language (ok, OSes are not currently powered by such languages, so that multiplexing is a system-level problem, with them!)

vii. Not seeing the importance of TRUST, I didn't at the time realize the effect of proprietary vs free software issues in the design of the system. Indeed, closed vs open source has a great impact on the dynamics of trust-building, on the need to have features and multiplexing in the ”kernel”; on the separation between users and programmers, etc, etc.

viii. About reuse and copy/paste: copy paste, of course, is evil. that's precisely why I cite it as the simplest and dumbest way. It's the way we use when better ways are available. We all use it a lot. Making other ways to reuse code difficult just makes us use this one, with all maintainability nightmare and development cost that this induces.
Of course there's even worse. In a language like unlambda (voluntary designed for obfuscation, yet based on nice theory), you mostly cannot even copy/paste code then insert modifications to do ”simple” things like add a variable (well, then reason you cannot, and how it generalizes to other languages is interesting, and deserves treatment). Not to talk about binary executable objects, that are typically a language where you have a hard time copy/pasting routines (a reason why we use assemblers and symbolic languages).

ix. Algorithms provide trust in the well-defined behaviour of a program: the systematic coverage of every case in some space, the strict abiding by rules known in advance, the controlled usage of space and time resources, are valuable meta-level knowledge about a program. “AI” techniques on the other hand, attempt to be somewhat creative, and hence unpredictable, and by definition, try to destroy any such

meta-level knowledge, although in most cases, we like to have some meta-meta-level knowledge about the goals that the AI seek, and some double-check on the fulfillment of these goals.
There are many cases such meta-level knowledge is necessary and where "AI" kind of techniques just cannot be used within a program, or can only be used with a algorithmic backup plan, when it is possible to provide such a plan (for instance, any somewhat real-time control problem might like to use AI to find interesting "optimized" solutions, but will require an algorithm guaranteed to give a sensible response in case the AI doesn't provide a satisfying answer in time).
However, just because a program's run-time must be somewhat algorithmic doesn't mean that AI cannot be used during development-time, compile-time, etc. Hopefully, AI can provide unpredictable creative help in generating predictable stubborn algorithms.

References

[1] Frédéric Bastiat. *The Law.* `http://bastiat.org/`, 1850.

[2] Frédéric Bastiat. *That Which is Seen, and That Which is Not Seen.* `http://bastiat.org/`, 1850.

[3] Gregory J. Chaitin. *The Limits of Mathematics.* Springer-Verlag, 1998.

[4] F. A. Hayek. *Law, Legislation and Liberty.* Routledge, 1982.

[5] Henry Hazlitt. *The Foundations of Morality.* 1964.

[6] John Stuart Mill. *Utilitarianism.* 1863.

[7] Norbert Wiener. *Cybernetics and Society.* Houghton Mifflin, 1957.