The Computer Nonsense Guide

Live for the Swarm! 10.03.2019

Abstract

"An object is really a function that has no name and that gets its argument a message and then look at that message and decide what to do next." — Richard P. Gabriel

This guide is product of the efforts of many people too numerous to list here and the unique environment of our open-source community.

The work presented here confines itself primarily to the stabler parts of the system, and does not address the window system, user interface or application programming interfaces at all.

We are an open-source research & development community that conducts multidisciplinary work on distributed systems, artificial intelligence and high-performance computing.

Our Mission: provide tools inside a simple workspace for play, work and science!

Our Goal: a distributed AI toolkit and workspace environment for machines of all ages.

We offer a custom Debian stable and a set of containers with native support for Python, Lua and Erlang ecosystems unified within our quasi-opportunistic blackboard; installation, consulting and support is also available from members of the community.

Core ideas

- Functions are a form of objects.
- Message passing and function calling are analogous.
- Asynchronous message passing is necessary for non-blocking systems.
- Selective receive allow to ignore messages uninteresting now.

Getting started

Please make sure that your Debian system have the latest release of Erlang, LuaJIT (with luarocks) and Singularity installed.

Installation

Then run this command:

```
luarocks install package
```

For help using the spacebeam package, including a list of commands, run:

```
$ pkg --help
```

Congratulations, you are jacked up and good to go!

Cognitive Constructivism

Piaget's theory of cognitive development proposes that humans cannot be given information, in which they immediately understand and use. Instead, learners must construct their own knowledge. They build their knowledge through experience. Experiences enable them to create mental models of the world. These models are changed, enlarged, and made more sophisticated through two complimentary processes: assimilation and accommodation.

Cognitive constructivism is based on two different senses of construction. First, on the idea that people learn by actively constructing new knowledge, not by having information poured into their heads. Moreover, constructivism asserts that people learn with particular effectiveness when they are engaged in constructing personally meaningful artifacts (e.g. computer programs, animations).

View of Learning

Because knowledge is actively constructed, learning is presented as a process of active discovery. The role of the instructor is not to drill knowledge into students through consistent repetition, or to goad them into learning through carefully employed rewards and punishments. Rather, the role of the teacher is to facilitate discovery by providing the necessary resources and by guiding learners as they attempt to assimilate new knowledge to old and to modify the old to accommodate the new.

What is an organization?

"[ORGs](#) is a paradigm in which people are tightly integrated with information technology that enables them to function with an organizationally relevant task or problem." — Carl Hewitt

A monkey, a building, a drone: each is a concrete object and can be easily identified. One difficulty attending the study of organizations is that an organization is not as readily visible or describable.

Exactly what is an organization such as a business concern? It is a building? A collection of machinery? A legal document containing a statement of incorporation? It is hardly likely to be any of these by itself. Rather, to describe an organization requires the consideration of a number of properties it possesses, thus gradually making clear, or at least clearer, that it is.

The purposes of the organization, whether it is formal or informal, are accomplished by a collection of members whose efforts or to use a term to be employed throughout this work, behavior are so directed that they become coordinated and integrated in order to attain sub-goals and objectives.

Perception and behavior

All of us at some point or another have had the experience of watching another person do something or behave in a certain way, saying to ourselves, "She/he acts as if she/he thought, ... " and then filling in some supposition about the way the other person looked at things.

Simple as the statement "He acts as if he thought ... " may be, it illustrates two important points.

First, what the person thinks he sees may not actually exist. They could act as if changes in methods as an attempt by management to exploit them.

As long as they had this attitude or belief, any action by management to change any work method would be met, at the very least, with suspicion and probably with hostility.

The second point is that people act on the basis of what they see. In understanding behavior, we must recognize that facts people do not perceive as meaningful usually will not influence their behavior, whereas the things they believe to be real, even though factually incorrect or nonexistent, will influence it.

Organizations are intended to bring about integrated behavior. Similar, or at least compatible, perceptions on the part of organizational members are therefore a matter of prime consideration.

Clues

One of the first things we must recognize is that in learning about things we not only learn what they are, that is, that the round white object is for football, but we also learn what these things mean, that is, football is a sport that the USA men's team don't get and their woman counterpart have master perfectly.

Upon receiving a signal (sight of football) we perform an interpretative step by which a meaning is attached to it.

Many of these "meanings" are so common and fundamental in our understanding of the world that we fail to note them except under unusual circumstances.

One way these meanings are brought home to us is by meeting people from countries different from our own; many of the meanings which things have come from our culture, they are things all people within the culture share.

These common interpretations of things help enormously in communicating, but they sometimes make it difficult to set factors in perspective so that we can really understand the reasons for behavior.

Threshold of perception

We all, have certain things (stimuli) to which we are sensitized and that when these appear we are instantly alert and eager to examine them.

There are other stimuli of relative unimportant to us to which we do not pay as much attention and may, in effect, actually block out.

One way of viewing this subject is to suggest that we have thresholds or barriers which regulate what information from the outside world reaches our consciousness.

On some matters the barriers are high and we remain oblivious to them, but on others which are quite important to us we are sensitized

and, in effect, we lower the barrier, permitting all the information possible concerning these matters to reach our consciousness.

Resonance

Related to this idea of sensitivity and selectivity is a phenomenon that might be called resonance.

Through experience and what we see ourselves to be, the understanding of a particular item of information may be very similar to that of others.

It is explained this way: since all the people inside a group look upon themselves as peers, they know what a change on the individual means in annoyance and inconvenience.

They can easily put themselves into his shows and, once having done so, probably feel almost as disturbed as he might be.

Internal consistency

One property of the images formed of the world around us is that they are reasonable, or internally consistent.

For instance, we may look at some draw on a page and see a rabbit. One portion along these lines might suggest a duck, but we do not have an image of something half rabbit and half duck.

In fact, if our first impression is of a duck, we may never notice that a portion looks like a rabbit.

We seem to tune out the elements that do not fit.

Dealing with conflict

Organizations that posses the capacity to deal adequately with conflict have been described as follows:

1. They posses the machinery to deal constructively with conflict. They have an structure which facilitates constructive interaction between individuals and work groups.

2. The personnel of the organization is skilled in the process of effective interaction and mutual influence (skills in group ledership and membership roles in group building and maintenance functions).

3. There is a high confidence and trust in one another among members of the organization, loyalty to the work group and to the organization, and high motivation to achieve the organization's objectives.

Confidence, loyalty, and cooperative motivation produce earnest, sincere, and determined efforts to find solutions to conflict. There is greater motivation to find constructive solution that to maintain an irreconcilable conflict. The solutions reached are often highly creative and represent a far better solution than any initially proposed by the conflicting interests.

The essence here is that out of conflict will come a new synthesis superior to what existed before and perhaps superior to any individual point of view existent in conflict.

Conflict, resting in part on different perspectives of what "ought" to be, is one of the avenues for opening new directions for the organization or one of the ways of moving in new directions. This is not only useful but also vital for organizational survival. The question, therefore, as we view conflict is not, "How to eliminate it?" but, "Is it conflict of such a type and within circumstances where it will contribute to rather than detract from organizational interest?"

Whether a conflict is good or bad for an organization, whether a conflict can be made useful for an organization, depends not so much on manipulating the conflict itself as on the underlying conditions of the overall organization. In this sense, conflict can be seen as;

1. a symptom of more basic problems which requires attention

2. an intervening variable in the overall organization to be considered, used, and maintained within certain useful boundaries.

Organizational adaptation frequently proceeds through a new arrangement developing informally, which, after proving its worth and becoming accepted, is formally adopted. The first informal development, however, may be contrary to previously established procedures and in a sense a violation or a subversion of them; or the informal procedures may be an extension of a function for internal political purposes.

Programmed links

If the process had given instruction to report immediately on completion of the task, this instruction facilitates linking the completed act with the next one. Through an information transfer, we call this a programmed link.

The supervisor node, of course may detect that something is wrong through another control cycle. It can then take corrective action by including or adding into this programmed link or perhaps by attacking on the more difficult problem of human apathetic attitudes and motivation his units could display.

Progression of goals

Organizations have progression of goals which result from a division of work.

A subdivide goal becomes the task of a process contained within a specialized organizational unit.

This nesting of goals is contained as part of the core organizational means-ends chain.

Needless to say, the hierarchy of control loops which are connected with the progression of goals may be handle in number of ways, regardless of how the elements are allocated, the important factor is that all elements must be provided for in some way. Hence, our model supplies an extremely useful tool in analyzing complex control situations by telling us what basic functions bust occur and in what sequence, even though initially we have no idea as to where or how they are executed in the organization.

Goals and feedback

The feedback loop containing information about organizational performance and conditions leads to definition of subunit goals or standards. It's important to show how a situation in one area could lead to modifications in a number of organizational units at higher levels.

This even result in reformulating the basic goals of organizations. Feedback is essential to adequate goal formation.

Why Erlang helps?

"Any sufficiently complicated concurrent program in another language contains an ad hoc informally-specified bug-ridden slow implementation of half of Erlang." — Robert Virding

Erlang suits iterative development perfectly, the ecosystem offers a variety of languages with different focus all build in top of the BEAM and the OTP framework.

Let it crash!

Robust systems must always be aware of errors but avoid the need of error checking code everywhere. We want to be able to handle processes crashes among cooperative processes.

- If one process crashes all cooperating processes should crash
- Cooperating processes are linked together
- Process crashes propagate along links

System processes can monitor them and rest them when necessary but sometimes we do need to handle errors locally.

Pattern matching

Functions use pattern matching to select clauses, this is a BIG WIN™

Supervision trees

This is where the supervisor process makes it entrance. It takes over the responsibility for the unexpected-error-handling and recovery strategies from the human developer.

The behavior, in a deterministic and consistent manner, handles monitoring, restart strategies, race conditions, and borderline cases most developers would not think of.

A supervisor has a standard set of interface functions and include functionality for tracing and error reporting. Supervisors are used to build a hierarchical process structure called a supervision tree, a nice way to structure a fault-tolerant application.

- Supervisors will monitor their processes through links and trapping exists.
- Supervisors can restart the workers when they terminate.

On production, this usually means a fairly straight-forward combination of external process management, overload monitoring and proxying.

A supervisor is responsible for starting, stopping, and monitoring external processes. The basic idea of a supervisor is that it is to keep its processes alive by restarting them when necessary.

Fault-tolerance

Fault-tolerance is achieved by creating supervision trees, where the supervisors are the nodes and the workers are the leaves of this analogy. Supervisors on a particular level monitor and handle children in the subtrees they have started.

If any worker terminates abnormally, the simple supervisor immediately restart it. If the process instead terminate normally, they are removed from the supervision tree and no further action is taken.

Stopping the supervisor results in all the processes in the tree being unconditionally terminated. When the supervisor terminates, the run-time ensures that all processes linked to it receive an EXIT signal.

It is a valid assumption that nothing abnormal should happen when starting your system. If a supervisor is unable to correctly start a process, it terminates all of its processes and aborts the startup procedure. While we are all for a resilient system that tries to recover from errors, startup failures is where we draw the line.

BEAM virtual machine

The virtual machine runs as one OS process. By default it runs one OS thread per core to achieve maximum utilization of the machine. The number of threads and on which cores they run can be set when the BEAM is started.

Erlang processes are implemented entirely by the VM and have no connection to either OS processes or OS threads. So even if you are running a BEAM system of over one million processes it is still only one OS process and one thread per core, in this sense the BEAM is a "process virtual machine" while the Erlang system itself very much behaves like an OS and Erlang processes have very similar properties to OS processes.

- Process isolation
- Asynchronous communication
- Error handling, introspection and monitoring
- Predefined set of datatypes
- Immutable data
- Pattern matching
- Functional, soft real-time, reactive, message-passing system
- Modules as function containers and the only way of handle code

Inside the BEAM ecosystem, we just worry about receiving messages.

Load balancing

The goal is to not overload any scheduler while using as little CPU as possible.

Compacting the load to fewer schedulers is usually better for memory locality, specially on hyperthreads, the primary process is in charge of balance the rest of the schedulers.

Process stealing

Process stealing is used by artists of all types and computers alike, on the BEAM is the primary mechanism to load balance and spread processes.

- A scheduler with nothing runnable will try to "steal processes" from adjacent schedulers, then next beyond that.
- We only steal from run-queues, never running or suspended processes.
- Schedulers changes on other schedulers run-queues.
- Each scheduler has its own run-queue.
- Processes suspend when waiting for messages, this is NOT a busy wait.
- Suspended processes become runnable when a message arrives.

By this mechanism the BEAM suspend unneeded schedulers. Once every period of 20k function calls is reach a new primary process inside a node scheduler is chosen.

Functions and modules

Modules contain functions, its a flat module space with just functions they only exist in modules there are no dependencies between running modules they can come and go as they please.

Functions

Functions cannot have a variable number of arguments! Erlang/OTP assumes functions with same name but different arities, each function has only a fixed number of arguments.

Modules

Modules can have functions with the same name and different number of arguments (arity), inside the virtual machine they are different functions.

Modules can consist of

- Declarations
- Function definitions
- Macro definitions
- Compile time function definitions

Macros can be defined anywhere, but must be defined before used.

The system only has compile code there is no build-in interpreter just compile code in modules. Everything is in modules the module is

the unit of code handling, you compile modules, load modules, delete modules, update modules, everything run though modules there are no living functions outside modules.

We can have multiple versions of modules in the system at the same time, all functions belong to a module, this handle of modules means there is no inter-module dependency of modules at all, they just come and go when the system is running.

In this sense a running BEAM instance has no notion of a system, and can be described more like a collection of running modules.

History of Lisp 3

"Lisp is the greatest single programming language ever designed." — Alan Kay

A lot has changed since 1958, even for Lisp it now has even more to offer.

- It's a programmable programming language
- As such, it's excellent language for exploratory programming.
- Due to it's venerable age, there is an enormous corpus of code and ideas to draw from.

Overall, the evolution of Lisp has been, guided more by institutional rivalry, one-upmanship, and the glee born of technical cleverness characteristic of the hacker culture than by sober assessment of technical requirements.

Lisp 1

Early thoughts about a language that eventually became Lisp started in 1956 when John McCarty attended the Dartmouth Summer Research Project on Artificial Intelligence.

The original idea was to produce a compiler, but in the 50's this was considered a major undertaking, and McCarthy and his team needed some experimenting in order to get good conventions for subroutine linking, stack handling and erasure.

They started by hand-compiling various functions into assembly language and writing subroutines to provide a LISP environment.

They decided on garbage collection in which storage is abandoned until the free storage list is exhausted, the storage accessible from program variables and the stack is marked, so the unmarked storage is made into a new free storage list.

At the time was also decided to use SAVE and UNSAVE routines that use a single contiguous public stack array to save the values of variables and subroutine return addresses in the implementation of recursive subroutines.

Another decision was to give up the prefix and tag parts of the message, this left us with a single type an 15 bit address, so that the language didn't require declarations.

These simplifications made Lisp into a way of describing computable functions much neater than the Turing machines or the general recursive definitions used in recursive function theory.

The fact that Turing machines constitute an awkward programming language doesn't much bother recursive function theorists, because they almost never have any reason to write particular recursive definitions since the theory concerns recursive functions in general.

Another way to show that Lisp was neater than Turing machines was to write a universal LISP function and show that it is briefer and more comprehensible than the description of a universal Turing Machine.

This refers to the Lisp function `eval(e,a)` which computes the value of a Lisp expression `e`, the second argument `a` being a list of assignments of values to variables, `a` is needed to make the recursion work.

Lisp 2

The Lisp 2 project was a concerted language that represented a radical departure from Lisp 1.5.

In contrast to most languages in which the language is first designed and then implemented Lisp 2 was an implementation in search of a language, in retrospect we can point out that was searching from one out of Sweden.

The earliest known LISP 2 document is a one-page agenda for a Lisp 2 Specifications Conference held by the Artificial Intelligence Group at Standford. Section 2 of this agenda was:

Proposals for Lisp 2.0

- Linear Free Storage
- Numbers and other full words
- Auxiliary Storage
- Input language, infix notation.
- Arrays
- Freer output format
- Sequence of implementation
- Comments

- Documentation and maintenance
- Hash Coding
- Sobroutine linkage
- Storage conventions
- Effect of various I/O apparatus
- Interaction with programs in other languages
- Expressions having property lists

The Actor Model

Actors are the universal primitive of concurrent digital computation. In response to a message that it receives, an actor can make local decisions, create more Actors, send more messages, and designate how to respond to the next message received.

Unbounded nondeterminism is the property that the amount of delay in servicing a request can become unbounded as a result of arbitration of contention for shared resources while still guaranteeing that the request will eventually be serviced.

Arguments for unbounded nondeterminism include the following:

There is no bound that can be placed on how long it takes a computational circuit called an Arbiter to settle.

- Arbiters are used in computers to deal with the circumstance that computer clocks operate asychoronously with input from outside, "e.g, keyboard input, disk access, network input, etc..."

- So it could take an unbounded time for a message to sent to a computer to be received and in the meantime the computer could traverse an unbounded number of states.

The following were the main influences on the development of the actor model of computation:

- The suggestion by Alan Kay that procedural embedding be extended to cover data structures in the context of our previous attempts to generalize the work by Church, Landin, Evans, and Reynolds on "functional data structures."
- The context of our previous attempts to clean up and generalize the work on coroutine control structures of Landin, Mitchell, Krutar, Balzer, Reynolds, Bobrow-Wegbreit, and Sussman.
- The influence of Seymour Papert's "little man" metaphor for computation in LOGO.
- The limitations and complexities of capability-based protection schemes. Every actor transmission is in effect an inter-domain call efficiently providing an intrinsic protection on actor machines.
- The experience developing previous generations of PLANNER. Essentially the whole PLANNER-71 language (together with some extensions) was implemented by Julian Davies in POP-2 at the University of Edinburgh.

In terms of the actor model of computation, control structure is simply a pattern of passing messages.

We have quoted Hewitt at length because the passage illustrates the many connections among different ideas floating around in the AI, Lisp, and other programming language communities; and because this particular point in the evolution of ideas represented a distillation that soon fed back quickly and powerfully into the evolution of Lisp itself.

Logic and λ-calculus

Logic programming is the proposal to implement systems using mathematical logic.

Perhaps the first published proposal to use mathematical logic for programming was John McCarthy's Advice Taker paper.

Planner was the first language to feature "procedural plans" that were called by "pattern-directed invocation" using "goals" and "assertions". A subset called Micro Planner was implemented by Gerry Sussman, Eugene Chariak and Terry Winograd and was used in Winograd's natural language understanding program SHRDLU, and some other projects.

This generated a great deal of excitement in the field of AI. It also generated controversy because it proposed an alternative to the logic approach one of the mainstay paradigms for AI.

The upshot is that the procedural approach has a different mathematical semantics based on the denotation semantics of the Actor model from the semantics of mathematical logic.

There were some surprising results from this research including that mathematical logic is incapable of implementing general concurrent computation even though it can implement sequential computation and some kinds of parallel computation including the lambda calculus.

Classical logic blows up in the face of inconsistent information that is kind of ubiquitous with the growth of the internet.

This change enables a new generation of systems that incorporate ideas from mathematical logic in their implementation, resulting on

some reincarnation of logic programming. But something is often transformed when reincarnated!

A limitation of logic programming

In his 1988 paper on early history of Prolog, Bob Kowalski published the thesis that "computation is controlled deduction" which he attributed to Pat Hayes.

Contrary to Kowalski and Hayes, Hewitt's thesis was that logical deduction was incapable of carrying out concurrent computation in open systems because of indeterminacy in the arrival order of messages.

Indeterminacy in concurrent computation

Hewitt and Agha [1991] argued that: The Actor model makes use of arbitration for determining which message is next in the arrival ordering of an Actor that is sent multiple messages concurrently.

For example Arbiters can be used in the implementation of the arrival ordering of an Actor which is subject to physical indeterminacy in the arrival order.

In concrete terms for Actor systems typically we cannot observe the details by which the arrival order of messages for an Actor is determined. Attempting to do so affects the results and can even push the indeterminacy elsewhere.

Instead of observing the internals of arbitration processes of Actor computations, we await outcomes.

Physical indeterminacy in arbiters produces indeterminacy in Actors. The reason that we await outcomes is that we have no alternative because of indeterminacy.

According to Chris Fuchs [2004], quantum physics is a theory whose terms refer predominately to our interface with the world. It is a theory not about observables, not beables, but about 'dingables' we tap a bell with our gentle touch and listen for its beautiful ring.

It is important to distinguish between indeterminacy in which factors outside the control of an information system are making decision and *choice* in which the information system has some control.

It is not sufficient to say that indeterminacy in Actor systems is due to unknown/unmodeled properties of the network infrastructure. The whole point of the appeal to quantum indeterminacy is to show that aspects of Actor systems can be unknowable and the participants can be entangled.

The concept that quantum mechanics forces us to give up is: the description of a system independent from the observer providing such a description; that is the concept of the absolute state of a system. I.e, there is no observer independent data at all.

According to Zurek [1982], "Properties of quantum systems have no absolute meaning. Rather they must always characterized with respect to other physical systems."

Does this mean that there is no relation whatsoever between views of different observers? Certainly not. According to Rovelli [1996] "It is possible to compare different views, but the process of comparison is always a physical interaction (and all physical interactions are quantum mechanical in nature)".

Lisp 3

Lisp Flavored Erlang (LFE) is a functional, concurrent, general-purpose programming language and Lisp dialect built on top of Core Erlang and the Erlang Virtual Machine (BEAM).

What isn't

- It isn't an implementation of Maclisp
- It isn't an implementation of Scheme
- It isn't an implementation of Common Lisp
- It isn't an implementation of Clojure

What is

- LFE is a proper Lisp based on the features and limitations of the Erlang VM (BEAM).
- LFE coexists seamlessly with vanilla Erlang/OTP and the rest of the BEAM ecosystem.
- LFE runs on the standard Erlang Virtual Machine (BEAM).

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in LFE and used by the

Spacebeam package system. Its purpose is to perform generic operations on objects. Part of its implementation is simply a convention in procedural-calling style: part is a powerful language feature, called flavors, for defining abstract objects.

Lisp Machine flavors

When writing a program, it is often convenient to model what the program does in term of objects, conceptual entities that can be likened to real-world things.

Choosing what objects to provide in a program is very important to the proper organization of the program.

In an object-oriented design, specifying what objects exist is the first task in designing the system.

In an electrical design system, the objects might be "resistors", "capacitors", "transistors", "wires", and "display windows".

After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it.

More rigorously, the program defines several types of object, and it can create many instances of each type.

The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of that type.

The new types may exist only in the programmer's mind. For example, it is possible to think of a disembodied property list as an abstract data type on which certain operations such as `get` and `put` are defined.

This type can be instantiated by evaluating this form you can create a new disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view.

However, such conceptual data types cannot be distinguished automatically be the system; one cannot ask "is this object a disembodied property list, as opposed to an ordinary list".

We represent our conceptual object by one structure.

The LFE flavors we use for the representation has structure and refers to other Lisp objects.

The object keeps track of an internal state which can be examined and altered by the operations available for that type of object, get examines the state of a property list, and put alters it.

We have seen the essence of object-oriented programming. A conceptual object is modeled by a single Lisp object, which bundles up some state information. For every type there is a set of operations that can be performed to examine or alter the object state.

Lua in Erlang

"Scripting is a relevant technique for any programmer's toolbox." — Roberto Ierusalimschy

[Luerl](#) is an implementation of standard Lua 5.3 written in Erlang/OTP.

Lua is a powerful, efficient, lightweight, embeddable scripting language common in games, IoT devices, machine learning and scientific computing research.

It supports procedural, object-oriented, functional, data-driven, reactive, organizational programming and data description.

Being an extension language, Lua has no notion of a "main" program: it works as a library embedded in a host. The host program can invoke functions to execute a piece of Lua code, can write and read Lua variables, and call Erlang functions by Lua code.

Luerl is a library, written in clean Erlang/OTP. For more information, check out the [get started](#) tutorial. You may want to browse the [examples](#) source code.

Luerl goal

A proper implementation of the Lua language

- It SHOULD look and behave the same as Lua 5.3
- It SHOULD include the Lua standard libraries
- It MUST interface well with Erlang

Embedded language

Lua is an embeddable language implemented as a library that offers a clear API for applications inside a register-based virtual machine.

This ability to be used as a library to extend an application is what makes Lua an extension language.

At the same time, a program that uses Lua can register new functions in the Luerl environment; such functions are implemented in Erlang (or another language) and can add facilities that cannot be written directly in Lua. This is what makes any Lua implementation an extensible language.

These two views of Lua (as extension language and as extensible language) correspond of two kinds of interaction between Erlang and Lua. In the first kind, Erlang has the control and Lua is the library. The Erlang code in this kind of interaction is what we call application code.

In the second kind, Lua has the control and Erlang is the library. Here, the Erlang code is called library code. Both application code and library code use the same API to communicate with Lua, the so called Luerl API.

Modules, Object Oriented programming and iterators need no extra features in the Lua API. They are all done with standard mechanisms for tables and first-class functions with lexical scope.

Exception handling and code load go the opposite way: primitives in the API are exported to Lua from the base system C, JIT, BEAM.

Lua implementations are based on the idea of closures, a closure represents the code of a function plus the environment where the function was defined.

Like with tables, Luerl itself uses functions for several important constructs in the language. The use of constructors based on functions helps to make the API simple and general.

The result

Luerl is a native Erlang implementation of standard Lua 5.3 written for the BEAM ecosystem.

- Easy for Erlang to call
- Easy for Lua to call Erlang
- Erlang concurrency model and error handling

Through the use of the BEAM languages, Luerl can be augmented to cope with a wide range of different domains, creating a customized language sharing a syntactical framework.

Process of abstraction

In trying to understand what is happening around us we are faced with a fundamental problem. In approaching any situation, the system trying to understand it, does not attempt to gather all information. Instead it selects certain facts and searchers for others.

This selection of some items and ignoring of others is a process of abstraction.

It is the abstracting form a real or if you will empirical situation the things seemingly most important to deal with.

In this process of abstraction and model building we deliberately select a few items, ignore may others, and then place the items chosen in a particular relationship to one another.

In doing so we are intentionally ignoring facts or relationships that can influence the type of situation under study.

The problem it to select the most meaningful elements and relationships and dropout the rest.

Those who use abstraction skillfully know well that they neither have all the facts nor have considered all the relationships bearing on the outcome of what they are analysing.

We do not use the abstractions from one situation in another setting without carefully examining the fit. Neither do we expect a model to handle all aspects of a situation.

We shall be dealing with many abstractions and models, not with the intention of exactly mirroring the real world but with the objective of clarifying our perception of its most essential features.

Abstractions and models are mechanisms for economizing both time and effort, but like any tool they must be used within their limits.

Model your goals

Taking the abstracted elements, a character with the flat tire begins to connect them into a pattern.

Better yet, he weaves them into a model of the confronting situation, which we can use both to understand his plight and figure out what to do about it.

The parts of this model would probably include, among other things, the flat tire, the image of the spare in the trunk, the telephone, the service station, a forthcoming business meting, etc.

A second model would contain the telephone, the service station, and the repairman there.

Finally, it concludes that it will call a cab and leave his wife to deal with the flat tire as best as she can.

These are extraordinarily elementary models, but they serve a very practical purpose.

With them the main character in our illustration can see the likely consequences of various courses of action.

We can find out these things by doing them directly by actually handling the tire and observing that we get dirty, or by calling the repairmen and waiting for him and learning that it takes too long.

In the age of big data; big models are good.

- For any given size of data, the bigger the model, the better it generalizes, provided you regularize well.
- This is obviously true if your model is an ensemble of smaller models.
- Adding extra models to the ensemble always helps.
- It is a good idea to try to make the data look small by using a big model.

By using the model, however, we can make some reasonable predictions about what will occur and therby accept or reject the choices open to us.

Predictive modeling

Our emphasis is on people in organizations where managers have to make decisions even through their knowledge and concepts may be wrong or inadequate.

As generalizations are made about what takes place in organizations, we shall often be talking about the decision processes that organizational members engage in or the sequence of factors which lead to a particular behavior.

We talk about the possible actions a person may take on recepipt of an order from his superior, which he thinks is improper and

unwarranted.

We talk about choossing amount the courses of accepting the order competly, protesting, protesting and raiisng an alternative, or leavng the organization.

It sometimes appers as if there were a basic assumption that people consicosly bring our all these possibilities and rationally weigh the pros and cons of each.

Does the individual at times carry on the same process unconsicusly that on other occasions he perhaps conducts consciously? We hardly know...

We are faced with the fact that people sometimes do things and later say, I never thougth that I would act like that under those circumstances.

In saying this the individual indicates that he saw other opportunities and that, through some process unobservable to him he decided among them and chose on that came from elsewhere that his own conscious thought.

The distinction between conscious and unconscious thought are by no means easy to determine, and for our purposes, it is not usually necessary to make them.

Considering the development of the field of Artificial Intelligence at the moment, it seems reasonable to conduct our analysis at the level at which both machines and humans do make decisions, without taking into account whether the choices and decisions processes are conscious or not.

Several references have been made with the intent of this guide to provide conceptual tools for analysis. As with any other tool models, abstractions and generalizations are useful only when within their limitations.