# Search for Effectiveness

Linas Vepstas

30 June 2025

**Abstract**

Personal letter containing extract from diary.

## Preface

Adam, This PDF is both a personal email and an extract from today's diary entry. Got home from my conference (Santara–Šviesa) and the below popped into my head while doing email chores. It touches on two things we talked about. Most important for you is perhaps the realization that any no–information–loss algorithm is a hill–climbing algo, and like any hill–climber, risks getting stuck at a local maximum. So this pours a little bit of cold water on hopefullness. But not all is lost; see below.

Rather than adapting the diary entry to you personally, its easier to just cut–n–paste. The first half for you will be entirely familiar and obvious; and as such I would ask that if you know of any papers that give more precise definitions and common notation, please send them to me. I'd rather not invent brand–new incompatible notation if that can be avoided.

Towards the middle I hit the hill–climbing dilemma. The meta–context is using graph generation and graph rewriting to explore monoids and the homorphisms (adjoint functors) between them, and to automatically find (generate) optimizations for space–time performance. See below for details. At the end is the use of "large structure models" (LSM's??) to correlate insn sequences, and the data they act on (in your demo, the Rubicks cube). The text might give some insight in why I want to tackle the general case.

Without further ado, the diary extract.

## Search for Effectiveness

You pointed me at the quad–store, the subtlety I had not previously appreciated. It is a row–column table, fixed at four columns. Two columns can be used for edge head and tail, and two more columns for "other things": edge colors, for example, but also graph labels, so that an indicated graph (all colored a single uniform color) can correspond to a vertex, and thus used to represent a certain form of hypergraph.

The structure of this table can be axiomatized. Using conventional axiomatics for directed graphs, requiring two columns, plus to (as of now) unspecified extra columns. From this, we can define a "Logic of Graphs", https://en.wikipedia.org/wiki/Logic_of_graphs or more precisely, a loose generalization thereof.

Keep in mind that I also want to axiomatize logic as jigsaws... but I digress.

Accompanying the axiomatization of the quad–store is an accounting of the space–time usage/performance of performing access. Part of the secret of the quad-store is that one has a collection of maps, and indexes for those maps (indexes providing rapid lookup/access, instead of brute–force search) that give all the different ways of currying four columns

$$(c_1, c_2, c_3, c_4) = c_1 \times c_2 \times c_3 \times c_4$$

which I attempt to tabulate:

| curried form |
|---|
| $\varnothing \to (c_1, c_2, c_3, c_4)$ |
| $c_1 \to (c_2, c_3, c_4)$ |
| $c_2 \to (c_1, c_3, c_4)$ |
| $\cdots$ |
| $(c_1, c_2) \to (c_3, c_4)$ |
| $\cdots$ |
| $(c_1, c_2, c_3) \to c_4$ |
| $\cdots$ |
| $(c_1, c_2, c_3, c_4) \to \varnothing$ |

which I guess forms a lattice(?) or I guess can be termed as the "language of a quad–store", where "language" is used in the sense of model theory. To do this, I have to be careful to write down the term algebra for the model: the variables, terms, predicates, relations that generate the language. And I'm too lazy to write them down write now, and perhaps they've been written down, by someone else, somewhere else. Adam had some notation for the above, presumably from such theoretical analyses.

Let's now change the topic to the axiomatization of computing. The oldest is, of course, the Turing machines, which axiomatize as a set of symbols, an infinite tape, and two transition functions: one that specifies the next state, and one that manipulates the tape. The observation here is that there are other possible axiomatizations. Perhaps the simplest is the counter machine https://en.wikipedia.org/wiki/Counter_machine, which have a handful of registers that can store arbitrary–precision integers.

Two remarkable things happen. First, the transition functions are now understood as "instructions", and algorithmic combinatorials of these can be understood as expanded instruction sets. Second, by the clever mapping of Godel numbers, the arbitrary precision numbers can be mapped to tape contents, and a demonstration of the Turing completeness of the counter machine can be derived.

What I want to focus on here is the role that combinatorial generation of the axioms play. That is, we have a bag of axioms, shake them about to generate a free monoid. That free monoid contains "all possible computer programs" (written in the language of the counter machine) and this monoid is exactly the same thing as the model–theoretic language generated by the syntax of the model. Elements of the monoid can be understood to be "abstract syntax trees"; this is the "shallow" statement that elements of a monoid are representable as trees.

The equivalence of the counter machine to the Turing machine proceeds via the Godel numbering, plus a clever insight as to cutting the tape in half to create to stack machines, plus one more cleverness to merge two stacks into one, etc. So we have a homomorphism (but not an isomorphism) of counter machines to Turing machines.

Assigning performance metrics to the execution of transition functions aka instructions, we see these have radically different performance profiles. This allows deriving theorems about PSPACE and whatever, to make claims about bounds and runtimes. There are also other models, the register machines https://en.wikipedia.org/wiki/Register_machine which include random–access machines and random–access stored–program machines, the latter being a model of present–day digital computers. These all have homomorphisms between each other. These all have different performance profiles.

The meta–perspective is ask about the automatic exploration of these equivalences, of the algorithms, of the optimizations and performance. The algorithms can be viewed in several ways. First, as abstract syntax trees. But also as instruction sequences represented as data–flow graphs. This is in the sense of gimple: an instruction has some input registers, some output registers, some clobbered registers; these are graph vertices, and the dependencies are graph edges. Compiler optimizers can be understood as graph–rewriting systems, recognizing an input subgraph, rewriting it into a more optimal subgraph, with better execution time, or perhaps fewer dependencies (thus, VLIW and/or pipeline stalls.) This is "existing" technology (much of which is extremely proprietary, and so hard to evaluate). But this existing technology is limited to the exploration of equivalences of instruction sequences for fixed architectures, whereas I am dreaming of exploration of the equivalences of structures across different architectures (with the most primitive example being the equivalence of Turing machines and counter machines.)

This exploration requires, at a minimum, the construction of an infrastructure in which axioms (model–theoretic terms, syntax) can be represented, and the resulting languages (trees) can be (freely) generated. This infrastructure also requires the ability to represent homomorphisms between different architectures. Some of these homomorphisms are "single shot" e.g. the mapping from Turing to counter machines, while other homomorphisms (isomorphisms??) are "multi–shot", a sequence of rewrites that may or may not be confluent, but in general are meant to performance–improving.

This immediately raises multiple issues. First and most obviously, it is known that graph rewriting systems are not, in general, confluent; sequences of rewrites

drive to different locations. Next is the issue of hill–climbing and local maxima. It is well–understood that different sequences of movements can lead to inescapable local maxima, and algos such as simulated annealing must be applied to escape such maxima.

This now raises a vexing insight. Adam presented a no–information–loss, strictly–uphill climber algorithm, using lattices with Galois connections on them. Brilliant! With one little issue: perhaps it climbs to a local information maximum, and gets trapped there. There are two ways to think about this situation. One is to imagine how great a tragedy this is, that one must first forget stuff (climb down the hill), in order to discover even greater knowledge. And one can, with some wistful emotions, come to terms with this. The other way is to look at this and say "A hah! This is exactly why the multiverse is needed!" That is, we explore all possibilities, "in parallel" or "simultaneously", as you wish, and then eventually abandon those paths which dead–end on a hilltop, and fail to be on the path to enlightenment. Of course, this now becomes a bit of an intractable problem, as there is a combinatorial explosion of multiverses to explore, and the pruning of dead ends seems to be infrequent. One can, again, wistfully sigh, but now with a different perspective.

If I follow my nose, I now ask: "what algorithms are best at pruning dead–ends?" which leads to another awkward situation: "how do I know I have reached a local maximum"? This question is "easy" if there is a finite number of possibilities to explore: they can be exhaustively tested. If this finite number is large, there's a problem. If the number of choices is infinite, exhaustive search cannot work, and so now one must depend on theorems that somehow reduce the infinite problem in some algebraic, proof–theoretic fashion, leaving us with only a few choices, each of which leads to a singular conclusion: the hill–top has been found.

Lets return now to the earlier desire to generate (enumerate) monoids. There are several ways in which this can be done. First is to crack open some book on model theory, take a close look at the formal definition of syntax, and to create a generator/enumerator according to that. Of course, this misses the whole point of Atomese, or rather, of jigsaws, which is the claim that jigsaws provide a better formalism for constructive generation, in that a partly–assembled (pre-)sheaf makes it clear what the unattached connectors are. More strongly, the sheaf axioms are more compact, and provide a certain scale independence lacking in other systems.

But is this an illusion? Well, sort–of. The jigsaws can, of course, be represented as lambdas. And lambdas, can, of course, be represented as combinators. And there are homomorphisms: there is a map from jigsaws to lambdas that is onto, and a map from lambdas to jigsaws that is into. Similarly, there is a map from jigsaws to term algebras that is onto, and a map from term algebras to jigsaws that is into. There is a map from jigsaws to quad–stores that is onto, and a map from quad–stores to jigsaws that is into. And likewise for SQL, and column stores, and so on. And presumably Adam's system, although the onto/into relationship is not yet known to me. Each of these different kinds of systems have distinct space and time performance. My claim has been that

jigsaws offer a superior space–time performance profile over all other systems; Adam's work has cast this into doubt.

And that's good, because, first, it highlights the nature of the claim, and second, it opens the door to automatic exploration of the homomorphic maps between these different systems. Adam's structure is apparently embarrassingly parallel for graph–rewriting. The current AtomSpace query engine is parallelizable, and has been parallelized, but no present–day applications particularly need to make use of this. Adam's system packs the data representation into cache–lines, for appropriate CPU performance. Is it portable to GPUs? Who knows.

Which raises another class of possible adjoint relationships. But first, an aside/footnote: I think that these onto–into maps between different systems should be properly called adjoint functors, although it will take a bit of work and precision to convince all readers that this is not an abuse of terminology.

Anyway, there are adjoint relationships between these systems and systems that describe parallelism. At the most base level, this is the pi calculus. At a slightly more general level, the assortment of process calculii. At the string level, there's the history monoid and the trace monoid, which describe partially–commutative monoids. If I recall correctly, these are Zariski topology frames and locales. And frames and locales are examples of the (I don't know what they are called, MacLane something–or–others) pieces making up topoi. So insofar as we wish to think of lambda–calculus as being single–threaded, the process calculii, the partially commutative monoids, the frames and locales, offer a collection of adjoint relationships between single–threaded and parallel systems. So we ask: what are the adjoint relationships that can be mapped to large abelian subalgebras, i.e. that can be highly parallelized? And even more interestingly, what adjoint maps can we discover to SIMD architectures, so that these can be moved to GPU's? I don't know how to take Atomese and make it SIMD; however, I do know how to write Atomese to search for adjoint functors between different axiomatic systems, some of which are parallelizable (MIMD) and some of which are SIMD-izable.

And again, the usual frustration: in principle, the usual theorem provers (e.g. HOL, Agda) should have been usable for these purposes, but are not. The intermediate languages in compilers, e.g. gcc gimple, or Microsoft's IR, can do this but are written to such a narrow and restrictive domain that they are all but unusable. In the middle of gimple sits a several register machine representations, and some homomorphisms between them, as needed to compile and optimize code. Somewhat similar remarks for the VLIW compilers, e.g. LVM clang for Qualcomm hexagon or the ARM Cortex etc. chips. There's nothing salvageable from these systems. Then there's the AtomSpace, where some pieces are robust and general purpose, while other pieces exist as prototypes, and the issues of the generation of jigsaws remain painfully underdeveloped. My gut instinct is to continue onwards with Atomese, rather than abandoning it and starting from scratch. The above paragraphs make it clear that, in the long run, the automatic exploration of adjoint relationships will strongly morph the system into something else. But that is in the mid–term future.

The project then requires several steps. First, an important step is to more closely review the conventional definitions of term algebras, and model theory syntax, and perhaps lambda calc, and write down explicit left and right adjoint functors between these, and jigsaws. Ditto for pi calc, and maybe some process calculus, and maybe for the history and trace monoids. The first few of these should be easy and straight–forward, if a bit verbose. Harder would be to then rephrase these results as frames and locales. At any rate, this lays out the formal groundwork needed so that others can understand what the heck this is about. I started this mapping many years ago, but was never motivated to finish it, because I did not see the "big picture" the way I do now. Hopefully those texts are not in a pathetic state and can be salvaged and completed (they are in the "sheaf" directory.)

Next, each of these adjoint functors probably should realized as working code. So far, I've taken three baby–step proof–of–concept mappings: one for lisp, one for prolog, and one for python; these are in directories under "storage". There's also a half–finished mapping to SQL in the atomspace–bridge git repo. All of these are seat–of–the–pants code, written from gut intuition rather than formal definitions. With some proper notation for the adjoint functors, and a way of talking about frames and locales, and a way for mapping to and from the logic of graphs, perhaps a more sold, stronger, and clearer framework can emerge.

But what is the point of this framework? To see what it looks like. But it also feels like a distraction from the main project, which is to automatically generate sentences from the language, and to automatically discover homorphisms (adjoints) to other kinds of systems, and to find confluent rewrites, and to perform hill–climbing to find more efficient, optimized algos, and specifically, to automatically find both representations and algos that run well in SIMD and MIMD architectures, ideally constrained to specific cache–line sizes, bus latencies, etc.

So what is the best, most efficient, easiest and most direct path from where I am today, to a system capable of implementing the above? What is the path of least resistance? What is the path of least distraction?

Two more topics I need to digest today: revisit the above in the light of sensori–motor agents, and to comment on the very interesting task of correlating functional descriptions, i.e. instruction sequences, with how these instruction sequences process data streams, and the hope/plan that some RNN or other neural net architecture can be trained to correlate the insn sequences with the data streams. And, as before, the point is that the data sequences are coming from sensory systems, and the insn sequences are applied to the data streams, and that there are two things that happen: a world model is constructed (presumably as a monad, i.e. the agent is a pipeline of monads) and that motor movements result from a similar–but–different insn stream. And to keep in mind that the point of the RNN is to find ... confluent rewrites, or to find optimized rewrites, or to (more generally) evaluate and maintain collections of insns homotopic under the Zariski topology. Or something like that; we are getting vaporous here. Because (1) its time for dinner, and (2) converting this

last paragraph into something more specific will require many hours, if not days or weeks.

## The End