Datalisp

Overview of design decisions

April 6, 2021

1 Hedging

In this paper I won't shy away from bold statements. This necessarily means that I am relying on informal concepts, I hope to make these concepts clear enough to be modeled mathematically.

None of these ideas are my own, I have no idea. This is just a general direction that I can see things moving into and my attempt at finding what their convergence would look like at the limit.

To that end datalisp is work that will forever be in progress. The concept of a "data interchange format" is clearly fundamental and hopefully understood by all. Any solution is therefore necessarily incomplete and must be designed for adaptation.

The largest inspiration and fundamental primitive is the idea of a "propagator network" (henceforth "propnet"). This is an idea heralded by Sussman (Scheme, Art of the Propagator, SICP, Designing Software for Flexibility), Arntzenius (created datafun which is the direct inspiration to this project along with statebox which is where I learned everything about concurrency) and Kmett (Haskell lens library, guanxi, various Haskell interpretations of category theory) as well as others.

We will start from fundamental assumptions and work towards an implementation.

2 Problem

Our first problem, as we've already touched on, is that (Informal $\rightarrow$ Formal) $\sim$ Void as noted by Alan Perlis in one of his epigrams.

The informal notions we want to reason about are things like "trust", "truth" as well as how to (recursively!) evaluate the basis for arguments about the relative worth of other evaluations. This kind of problem statement makes my head hurt so let us formulate our goals as a series of questions:

- How much do we trust our systems?
- How much do we trust the information we receive?
- How much do we trust the people we communicate with?
- How much do we trust that they are who they say they are?
- How much do we trust that they said what we think they said?
- Ad infinitum.

- How do we make sure we aren't misunderstood?
- What can we assume when evaluating answers to these questions?

- How can we increase our confidence?

These questions are clearly bordering on the religious, so claiming that a formal argument can solve these problems once and for all is (practically) certainly a delusion.

Our goal is to have a clear premise and a realistic approach, albeit somewhat expedient at times.

3 Assumptions

We believe in a shared reality.

We believe decisions are made based on observation.

We believe observation can be modeled as a space of possibility.

We believe in *symmetry*; just as we observe, we are observed.

We define *information* as a better prediction of observation than the uniform random probability distribution over the space of possibility.

We define *behaviour* as continuous observations.

We believe in *balance*, therefore; Behaviour $\sim$ Information.

We claim that Behaviour = Trust(Information) can model any *entity* in the shared reality.

We believe in *freedom*; Trust $\subset$ Information $\times$ Behaviour.

We believe in *security*; observed information is monotone increasing.

We believe in *knowledge*; there is a hierarchy of desired behaviour.

We believe in *intelligence*; throughput $\sim$ latency^{-1} i.e. tradeoff between efficiency and redundancy.

We believe in *harmony*; there is a convergence of knowledge, $\exists$polite $\in \bigvee$ Knowledge.

We believe in *prosperity*; $\exists$world $\in$ Freedom $\wedge$ Harmony

4 Definitions

Signal vs Noise

Signal is information, that when trusted, probably improves behaviour. That is; it is information that ostensibly increases knowledge (when we believe it). Noise is the dual notion.

Graph

A *graph* is a structure defined Graph = (Vertex, Edge) by a collection of vertices and edges (that can be directed, undirected, many-to-many; i.e. "hyper", transitive, and lots of other things).

The general intuition behind graphs is that they are dots (vertices) and lines that connect the dots (arrows/edges). They can have all sorts of properties, notably; directed, acyclic, connected, etc.

One class of graph that is very useful is a *tree* (as well as a multitree), this is a connected graph where there is a unique path between any two vertices (and some other equivalent conditions).

We can find "largest spanning tree"s in graphs relatively quickly and we can "overlay" graphs by joining their edge sets together.

There are several representations of graphs available and they have different tradeoffs. Since we are a data interchange format, we care about "canonical decomposition into trees", this is because trees are easy to serialize compactly and canonically. The graph can be reconstructed by overlaying the trees on one another.

Lattice

A *lattice* $\mathcal{L}$ is an algebraic structure that can be defined in various (more-or-less) equivalent ways, we at least require a space of possibility and an operation called a "join" (respectively; "meet") that is:

$a \vee b = b \vee a$	Commutative	$a \wedge b = b \wedge a$
$a \vee (b \vee c) = (a \vee b) \vee c$	Associative	$a \wedge (b \wedge c) = (a \wedge b) \wedge c$
$a \vee a = a$	Idempotent	$a \wedge a = a$
$\exists \bot \in \mathcal{L}\ \forall a \in \mathcal{L}.\ \bot \vee a = a$	Bottom / Top	$\exists \top \in \mathcal{L}\ \forall a \in \mathcal{L}.\ \top \wedge a = a$

A *semi*lattice only has one of the operations. If a lattice has both operations then it is *distributive* if one operation distributes over the other, like so:

$$\forall x, y, z \in \mathcal{L}.\ x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$$

Furthermore, we have an ordering or implicit hierarchy (also in semilattices): $a \wedge b = a \Rightarrow a \leq b$.

Properties

For us the most interesting properties of lattices are to do with composition:

We can compose lattices *sequentially* by identifying the top of one with the bottom of the other; $\top_1 :=: \bot_2$.

We can have mutually independent lattices by creating a new top and a new bottom e.g. $\top := \top_1 \vee \top_2$.

Most importantly we can replace a vertex with a lattice or a sublattice with a vertex and the result is still a lattice. I call this the "zoom-lemmas" (no relation to the malware), it refers to our ability to zoom-into a vertex to see the itemized breakdown of the signal-calculations happening or zoom-outof a lattice that is irrelevant to our current investigation.

We can think of these lemmas as files and folders in our knowledge graph; no longer a filesystem-tree, but rather, a max-signal lattice constructed from many (perhaps contradictory) perspectives.

Most strict lattices that we use in computer science will have the top implicit, its meaning is *contradiction*, it *covers* the actual tops that each represent a *perspective* on the bottom.

Likewise we have an implied bottom at all times which is "void of information" it has less information than anything else. Hence it is the bottom.

The idea is to converge our perspectives and solve contradictions, thereby eliminating disputes, growing mutual trust and building higher tops.

Examples

A tree is a join semilattice.

A directed acyclic graph is not a lattice in the general sense since there can be ambiguity about how to resolve contradictions (if two uncomparable things seek resolution, the lattice will give a unique result while the DAG may suggest several). In datalisp we cheat and use probability weights to give an ordering of preference to a DAG (but only when we get in trouble, this kind of ordering is *subjective*, not *canonical*), the ordering may differ between places due to the different (often incompatible) assumptions that people take for granted.

A powerset which has union and intersection as the join and the meet (respectively) is the canonical example of a complete distributive lattice.

Any total order with max and min is a lattice.

Greatest common divisor and least common multiple is a lattice w.r.t. division in the natural numbers.

Boolean logic is a distributive lattice.

Propnet

A *propagator network* is a bipartite graph (hypergraph) where one partition of vertices is called "places" and the other partition of vertices (hyperedges) is (are) called "transitions".

Furthermore; the places must be (joinsemi)lattices and the transitions must be monotone functions.

Properties

A propnet is tangent to network conditions. It's a fully CP-system, that is; consistent and partition-tolerant. In the "intelligence tradeoff" we've chosen security over safety (the latter being a more opportunistic notion).

Furthermore; a propnet can schedule the propagation of information nondeterministically and still have deterministic behaviour (at the limit). This means we can loosely behave in a client-server model, meaning we are lazy about propagation and only respond to people who send us information they thought we didn't know (reciprocity).

Example

The internet is a propnet where each place is a max-signal lattice (defined in terms of the local notion of signal/noise, i.e. particular to each place).

The monotone functions are simply messages that could contain anything. In the worst case they are simply ignored and the signal to noise ration in the place stays the same. At best the information is worth keeping (even if it means you need to make space by deleting information with less worth).

It is simple to see then (using our understanding of lattices). That we can zoom into this max-signal lattice and elaborate the structure.

The transitions can be thought of as the fully automated routing algorithms (irregardless of whether they do any inferences / modifications on the messages being routed).

Politics

We define *politics* as "many acting as one", the process of performing politics is called *consensus* and it allows us to resolve conflicting points of view.

An important problem in this class is called *economics*, this is a process that tries to form consensus on *value*, that is; answer the question "what is valuable?", we call the resulting (abstract) notion; *worth*, and we say that *price* (a concrete notion) will converge on the worth.

When we speak of *evaluation* in an informal setting then we are referring to the process of forming consensus on value, that is; determining worth or trust*worth*iness.

Properties

There are two main properties to consider; whether we can align individual incentives with that of the group, and whether all information available is used when forming the consensus, ideally we manage to make decisions using all available information.

The former problem is called *tragedy of the commons* and we call systems that cannot solve this problem *ephemeral* or *unsustainable*.

The latter problem is called *byzantine fault tolerance* or *censorship resistance* and we call systems that cannot solve this problem *centralized*.

A sustainable decentralized system can solve both problems.

Examples

Bitcoin is the first theoretically byzantine fault tolerant computer system, however, due to it's immense energy usage it looks like it is nevertheless unsustainable (the incentives of the whole; preservation of the planet, do not align with individual incentives; create waste heat to secure the network and get paid).
If there was an external political situation that would limit the amount of mining to sufficient levels to transact while maintaining even distribution of power (for some definition of even that bitcoin is unable to motivate), then things could be more sustainable (no one would want to set off the arms-race, i.e. break the equilibrium of power, for fear of retaliation). Therefore, any system that is able to shape the world around bitcoin, in such a way that bitcoin is sustainable, also obviates the need for bitcoin.

Democracy (as implemented today) is another unsustainable system, each voter has such a small influence that it is rational to be ignorant and spend your time on other concerns. However, if everyone is ignorant then democracy ceases to function.

Prediction markets are an incentive mechanism which unifies the most rational strategy for the individual with the most beneficial behaviour (to the whole), that is: sharing your true beliefs.

A "trusted third party" is a notion that is not byzantine fault tolerant, any time you rely on someone or something being truthful without balancing that perception with every other information you have, then

you have censored information that could have influenced your verdict (however slightly).

A "constant function market maker" is a way to do currency conversion, due to convexity it aligns incentives (a local optima is also a global optima) and so long as it can operate in a decentralized setting (where there is no central source of truth that can answer "what is valuable?") then we can use this method of composition that preserves our desired properties and allows us to construct arbitrarily large political systems from smaller functioning parts..

Currency

A currency is a type of political system. Any one political system can be simulated by a currency, the distribution of tokens is decided by consensus.

For currencies we usually require a proof of eligibility to participate in the consensus. The choice of proof is often used to classify such systems.

Currency allows us to refine our understanding of a "byzantine fault tolerant system", namely we now have accounting, i.e. maintaining the invariant that no money is created (or destroyed) without consensus. This is first-order byzantine fault tolerance (named after first-order logic, which is complete) since we only need censorship resistant computation in this model it suffices to make computers do proof of work.

Second-order byzantine fault tolerance is the ability to resolve human disputes about facts (wikipedia edit wars). By the "impossibility of moving from the informal to the formal"-hypothesis, we can conclude that this form of fault tolerance is necessarily incomplete (there will be "good" things we won't be able to form consensus about). Therefore, humans will have to be the "proof of work" but the problem is not of doing work but rather determining how valuable the work being done is (in relation to all other work).

The *second-order decentralization* proposed in the paper is built on a "proof of trust" that we cannot fully specify but will attempt to make clear throughout.

Examples

Democracy uses *proof of identity* where the identity is defined by a central power and the holders of identity get one token every epoch that they then spend on representatives that hold influence with the central entity. This scheme gives disproportionate influence to the representative and is unable to preserve the intentions of the voters.

Bitcoin uses *proof of work* where the work is a timelock puzzle without a winning strategy, the game is therefore symmetric for all participants but the system is incapable of doing I/O with the world so it cannot account for externalities (such as rampant energy usage).

Various cryptocurrency systems use *proof of stake* which is vulnerable to the tragedy of the commons via coordinated bribery attacks.

The problem of deciding whether or not to apply a patch to a codebase in an adversarial setting is a "second-order decentralization"-complete problem. If we can do that then the codebase may as well be a ledger keeping track of account state (first-order) or the laws of a country (second-order, pun intended, refers to taxes).

Authoritarianism is a system where there is *proof of authority* and the one who has this proof has the

only token.

Fundamental Property

Currency can be exchanged. It is the means by which we compose political systems.

5 Model

We seek now to model reality in these terms. First we must consider representation, if two equivalent concepts look completely different then we get the proliferation of noise as our identification methods are less effective. Nowadays it is popular to throw money at the problem and "train" statistical models for recognizing patterns in very large amounts of data.

For a data interchange format intended to be used in a network that makes no prior assumptions about "sources of truth", a system that considers everyone equal a priori, there is no general way to obtain this data. This is known as "the data availability problem" and the fact that this problem hasn't been much of a problem (bittorrent, blockchain archives) is an example of "charitable behaviour" such behaviour could also be called "religious" since it does not naively maximize for short-term incentives and contributes to a certain "distortion of reality" (where the definition of "reality" is depending heavily on the unreasonable effectiveness of mathematics).

A more sustainable model is one where every participant in the network is able to make their own inference without having "the big picture"

Desires

We need our concrete representation of data to be self-describing, so that others know what to do with it; amenable to content-addressing, so that others can refer to it without ambiguity; semantically clear, so that we can reason about equivalencies; and built on cryptography and mechanism designs, so that we can achieve prosperity.

Knowledge

To evaluate the signal of some message, we could do a reverse lookup of its content-address, that is; ask the network for metadata about this message.

This motivates the notion of a "metadata lattice" which is one of the fundamental constructions in datalisp. Similar to the hyperlink in http.

We want to be able to use probabilistic logic programming to make inferences about the information geometry of the propnet. This sounds like word-salad (and to be fair it probably is, but it represents my best faith in how to approach this problem at the time of writing).

Information geometry is using differential geometry (in reverse) to uncover the geometric structure that is most likely to be "real" given your observations / the information available to you.

(Probabilistic) logic programming is a declerative way of traversing spaces, you can make a query and the evaluation should be able to tell you how much confidence the system has in whether (perpetually preliminary) results are factual. You can go to lichess and look at stockfish updating its predictions (for what is the best move in the given situation) in real time to get an idea of the UX in such a system.

Economics

The metadata lattice is built by considering some public information immutable and building references to it that then also become shared context.

Other *lattice types* may have stronger or weaker assumptions about structure, for example pijul is a version control system where patches (which are able to transform any file into any other file) form a CRDT (a lattice type), while vector clocks are a much more constrained construction made to decide whether some information has reached everyone in the network (yet).

We will describe the metadata lattice construction in greater detail later in this document. In this section we will discuss the tragedy of the commons and how to incentivice people to collaborate in the representation of metadata (again; to maintain canonicality and make equivalency more apparent).

By forming a local consensus (here referring to a "neighbourhood" not necessarily a single entity, but maybe a "small political actor") around the use of idioms in this "echo chamber" we can have words with clear meaning. This enables us to collect evidence for equivalencies with other such dialects (translations), this way we can gradually resolve contradictions (merge conflicts) and over time converge the dialects (of course with the aim of achieving confluence).

The consensus will also be discussed a bit later on. It is clearly the hardest problem that I will keep dancing around for the rest of the document, although I do have concrete ideas about how to realize it.. I am not an advanced enough practitioner of mathematics to evaluate whether the methods I want to use are fully applicable to the problem.

In this section we are not crossing any trust boundaries and are just considering what kind of proposals for idioms we should make to our peers in order to gain their favour (good proposals lead to favorable reception and allow us to earn respect).

Therefore, the "economics" we do locally are all about estimating the relative worth of external information sources. Through such estimations we can form beliefs about the viability of different proposals.

Politics

The first thing to consider is what happens when you can assume that a group of people are all talking about the same subject (some content-address).

In this case we can treat this subject as the bottom of a metadata lattice.

The way to cross trust boundaries is to either have evaluation of information finalized by a human approval / feedback (in the form of notifications or putting it in relevant newsfeeds, ordered by confidence) OR use cryptography to prove some structural properties.

The most important properties to prove are the lattice properties, because by the zoom-lemmas and recursive SNARKs we can turn our metadata lattices into what is known as "proof-carrying data" or *network types* in the datalisp parlance.

Without checking exhaustively that the combined structure is still a lattice we can construct a proof for our new layer in the structure by checking the proof associated with the structures it depends on and proving that we are constructing the new layer from those parts in a sound way (zooming, summing or sequencing

lattices). If the new layer was constructed correctly (trivial constructions, so small proof circuits) then we know that the whole structure is still a lattice!

Using these recursive succinct arguments of knowledge we have formed proof-carrying data ("network types") that can cross trust boundaries.

Ideally the politics involved in trust calculations should have the properties of prediction markets (where honesty is preserved and we have censorship resistance due to the economic incentives of betting when information you are confident in indicates ROI).

This will be expanded on in the section about consensus.

Internet

At the top level all we can say is who we are and how much we trust the thing we are broadcasting.

One layer down we had network types that allowed us to move things automatically across trust boundaries. Below that we had the economics local to a place (governed by a human that is opinionated on what information is worthy of their attention). Below that we had the concrete datastructure which we imbue with these semantics.

However, at this top layer is where the "governing human" is actually interacting with the system. We simply cannot avoid asking the human for help.

Therefore we endeavour to do it in the absolute least intrusive way possible. By taking messages on the internet, verifying who sent them and multiplying our prior perception of them (in the given context) with the confidence they had in what they were saying, we may order data probabilistically into "newsfeeds" that are not manipulated by external actors, yet theoretically as good a guess as we can make w.r.t. the ordering.

Of course the human is given a user interface that allows them to correct our misconceptions succinctly. This will be expanded on below.

The whole system is a prediction market on whether or not something is "fake news" (or which version of wikipedia should be shown to this particular user, not decided centrally by the "service provider" but locally by the human herself).

6 Implementation

We seek to have certain properties from our implementation, such as; human legibility, support for binary formats, simplicity of grammar and lack of corner cases.

Thankfully this is pretty much a solved problem. We know that regular languages are not expressive enough (semantically) and we know that context-sensitive languages are more complicated than necessary (syntactically). Therefore we are left with context-free grammars, the simplest of which is the Dyck-language; the balanced paranthesis language.

As for data within these delimiters, it can be denoted using prefix-length tokens, that is; varints followed by a delimiter and then the (precise) amount of data specified by the varint.

Note that the five objectives listed here form "a chinese philosophy", this pattern is omnipresent so I will briefly explain it (although it is a tangent it is a fun one).

- Convergence is like curiosity (always trying to fit everything together),
- Confluence is like desire (the holy grail property that is always out of reach)
- Consensus is like violence (glossing over the minority opinion and acting as if you may decide things for others)
- Canonicality is like deception (you think that you've found the right way to factor meaning until you realize that you haven't)
- Compactness is like vulnerability (you have no place to hide, no obfuscation or verbosity to conceil the fact that you don't know what you are talking about)

The idea is that Curiosity generates Desire which causes Violence which leads to Deception which crumbles into Vulnerability which gives you the ability to be Curious again. Furthermore; Curiosity can discover the Deception, Deception can deceive the one bound by Desire, Desire can obtain the one who is Vulnerable, Vulnerability can rob Violence of meaning, Violence can constrain Curiosity. This pentagram for generating relations and pentagon for overcoming relations is an omnipresent pattern in philosophy and most often identified with the chinese philosophy of the five elements.

I normally extend the system to include duality in the final step before you complete the cycle. Vulnerability can be self-imposed or forced, similarily we can have come full circle to discover that we overoptimised for compactness and what we actually wanted was completeness (or vice versa). The general idea is whether we are going closer to void or to contradiction (did we meet or did we join).

This little tidbit of philosophy was not intended, I just noticed that what I had written down just now fits it.

Canonical

What I have described is canonical S-expressions, they look like this:

```
(4:this2:is1:a4:list2:of6:tokens44:while this is one token containing 44 bytes.)
```

The reason they are "canonical" is because there is no ambiguity in presentation, we cannot insert whitespace in between tokens without getting a syntax error or considering those bytes as parts of tokens (increasing the prefix number to cover the extra bytes).

We are willing to go to extreme lengths to preserve this property, whether it is inventing homotopy type theory or thinking really hard about how to best structure our data ... the bottomline is that we want as much deduplication as possible! That way we can take full advantage of content addressing and have better anonymity, collaboration and clarity of thought.

Complete

We want each *form* in such an expression to be self-contained, the first *token* will always unify with a known *operator* that along with its *version* describes fully what to expect from and how to interpret the rest of the data in the form.

These self-describing structures should converge on having no ambiguity as they are immutable and backwards compatibility is first class.

```
(3:txt1:062:these bytes will be interpreted as ascii but version 1 as utf8)
```

is a *word* where the operator is "txt" and the version is "0". This will have meaning (as) globally (as you are aware exists); this word will have associated metadata with a pijul patch that implements support for the word in the implementation that it associates with. The metadata is a join of the codebase that needs support for the word and the word itself (or rather the patch that it associates with).

When peers exchange data they can also tell each other about patches that make software able to handle the data. Then they can crowdsource code reviews to find out how much confidence they can have have in the patches (I believe you are beginning to see how the consensus will work).

Furthermore, if the meet across the network is no longer the bottom of the codebase but rather the bottom AND some patches applied to it, then there is now no harm in moving the bottom up and collapsing that metadata into it.

Convergence

Convergence is implemented by having every operator versioned with a natural number. We can never run out of natural numbers so the global namespace is infinitely big (for each operator!). The *word* denoted by the (operator, version) pair can be increasingly well documented as more metadata refers to the content address of its definition.

Each non-monotonic change must be made through a meet-consensus (moving the bottom up), while applying patches can be a bit more promiscuous (moving a top, local to your computer, one step up by merging a patch). The join-consensus can be configured to have different tolerance bars depending on the place being patched. Some users are experts and review code themselves, others review up to their level of confidence, yet others are completely reliant on their transitive trust not being misplaced.

When we lose our "security assumption" and are in a situation where communication can consume our information, then we have a structure that is more general than a propnet. It is known as a "petri net" (sigma net / pre net are other categories in this family) but such nets are not necessarily fully CP (it's like a propnet with quantitative type theory, see: linear logic programming).

At the limit we want to eliminate the need for gratuitous redundancy and choose the most intelligent tradeoff possible for evaluating knowledge.

To do this we need a way to resolve conflicts when one word is associated with several (at least two) incompatible implementations (that somehow all manage to clear the trust threshold needed to merge the patch) for this problem we use a modified version of the snow* consensus protocols pioneered by the the avalanche cryptocurrency network.

These consensus mechanisms are known as "stochastic gossip consensus" because we form consensus by taking a sample of peers in the network and asking them which patch we should merge. Whatever happens we decide to agree with the majority, now others are asking the same question of random samples and each person will ask enough times so that they can be almost certain that every honest node in the network has come to the same conclusion as them (through the majority *probably* growing each time the game is played until there is a vanishing likelihood that an honest player would dispute the consensus).

In the datalisp world this consensus mechanism is continuously running lazily via metadata and only as much consensus as is strictly necessary is formed at each point in spacetime. This means we don't have

global consistency except eventually.

Confluence

In programming language theory there is a "holy grail" property that is called "confluence" (I think HoTT could be considered a confluent programming language?).

It means that you can have *extensional equality* in your programming language. That is; two things are equal once reduced to normal form if and only if they have the same meaning in the given context. That is; if they will induce the same behaviour (even if they are implemented in competely different ways, unless of course your extensional equality happens in a domain that cares about such implementation differences).

The goal of datalisp is to converge on a fully confluent language. Using consensus to prune unnecessary baggage from the working vocabulary and censorship resistance to make sure every meaning is as available as it is deemed valuable.

Compact

Our representation of different structures should be as compact as possible in the computational sense (how hard is it to traverse / parse), the cognitive sense (how hard is it to understand the construction) as well as the concrete sense (how much memory does it take to store the data).

We are always willing to pay the overhead of the operator and the version so that we know what data we are working with.

To this end we present some preliminary definitions of "metadata lattice" words in datalisp (i.e. under the *top* operator).

The big idea is to take advantage of spanning trees to form canonical decompositions of graphs and then represent these index structures as compactly as possible (so that we again get canonicality).

```
(3:map1:0xx:bitmap1yy:bitmap2zz:bitmap3)
```

denotes a mapping that is supposed to be used in a topology.

The mapping is a graph with properties defined by the word, the graph is constructed from vertices that are natural numbers (so also indices) using a prefix structure:

```
If the first bit is a 0 then you have a varint that is an indice.
If the first bit is a 1 then you have a varint that says how many children are in the form.
```

This way we construct super compact trees in a similar way to how the canonical S-expressions work (prefixes and varints).

The actual topology will start with a list of referenced information, here the available types are tokens, each wrapping a full canonical S-expression (henceforth csexp), or a reference to some content address that can be looked up and then placed as a token wrapping a full csexp.

The referenced data is sorted by treating the bytes as digits in a base256 number. The indices in the associated bitmaps refer to the sorted data. Therefore the top word will rely on both data and bitmaps and the actual vertices are data and the graph invariants can be calculated quickly as you overlay the edges from each tree onto the vertices from the first spanning tree.

The canonical decomposition of a graph can start by sorting the vertices into indices and then taking a spanning tree (or multitree) from the first vertex that has an edge. We can either have a lot of duplicated edges (taking spanning (multi)trees from each node in the graph as a seperate bitmap) or go down the list collecting the largest spanning trees from the remaining edges.

We can have several bitmaps on the same vertices and give them different labels, this can be compacted with another "top" version.

Consensus

The rough idea is that each person can locally decide to trust whatever and whoever they want. They can run any code, modify any program and communicate in any way.

They are incentiviced to communicate in a *trustworthy* way by using a construction unique to this project that I am calling "proof of trust"; based on the confidence people have in the information they have digested and the confidence other people have in those people it becomes possible to converge these observations (using bayes rule) into a common posterior.

This resembles a prediction market in the sense that you are incentiviced to make the most honest prediction you can about the signal-to-noise ratio of a certain piece of information that you have finished evaluating.

The gritty details may involve factor graphs, expectation propagation and adjustments to fit the "datalisp way" (convergence, canonicality, consensus, confluence and compactness)

The argument for how we are solving the tragedy of the commons in a byzantine fault tolerant way is based on the types. Internet level communication relies on local ranking of sources of information. Network types allow us to transport structures across trust boundaries but joining content is largely up to humans, the system tries to evaluate which join-requests are worth looking at by using the ranking of sources as a baseline. Trust allows us to move faster and join without scrutiny, because people self-evaluate their confidence in utterances and we can evaluate how much to trust such self-evaluations.

This is why we treat lattice types differently from network types. Small communities will be able to move fast while they have a shared vision (a commons) as arguments happen the community will fracture but from the beginning we always have the implicit bottom of Void and implicit top of Contradiction, since computers are constructs the fracture in a community with a shared base will result in a maintenance burden for each side of any argument. Thankfully the meet is made explicit and so it is quite easy for adversaries to keep collaborating (since they aren't the only people in the world the nash equilibrium is to "raise the tide" and build up the commons).

7 Algorithms

In order to achieve this there is a whole lot of algorithms involved.

Expectation Propagation

Trueskill by microsoft uses a construction based on "factor graphs" to calculate shared posteriors for independent evaluators. The algorithm is not guaranteed to converge but there is a paper from deepmind in 2017 about "stochastic natural gradient expectation propagation" that is guaranteed to converge.

The reason this is exciting is that factor graphs are also bipartite graphs / hypergraphs (just like propnets and petri nets) and they encode the "information geometry" of the structure (as opposed to the structural properties of the components).

Routing

Datalog (the precursor of datafun which is the inspiration for the semantics of datalisp) has:

- Naive evaluation: Where the list of inferences is run on the list of facts to produce new facts. Then you repeat without caching.
- Semi-naive evaluation: Now you cache, so you don't repeatedly make the same inferences in the same way (you can make the same inferences in different ways however).
- Bayesian semi-naive evaluation (not a real thing, just something I thought of): Rather than not using facts that you've already used (as in semi-naive evaluation) you now also exclude facts (or inferences) that you do not have enough confidence in (posterior couldn't meet threshold value).

The evaluation semantics of datalog coincide with the shortest vector path routing algorithms used to run the internet. By using datalisp to describe how to communicate in the network we can get closer to a goal of the system, which is to optimize the firing sequences in the world-wide propnet by calculating paths using the bayesian semi-naive evaluation or something else that allows us to be as lazy as possible without compromising security.

Hopefully these semantics are also close in meaning to the ones alluded to in the SNEP model (with which I am not yet familiar).

Oracles

Amoveo (also a large inspiration for this project) leans heavily into the properties of prediction markets as well as constant function market makers to try to create the tools necessary to account for externalities. It is so far the only somewhat realistic cryptocurrency system for decentralized governance.

Datalisp takes the opposite approach to most cryptocurrencies by focusing on the second-order decentralization almost exclusively, being less worried about completeness than coverage. A data interchange language must be able to cover any kind of communication usecase.

Communication

Proof-Carrying Data

We can wrap any csexp in a "verified" keyword that associates a tree index (made via DFS for example) with a proof that the word for that form is respected by the data in this particular content-address.

This way we can keep the inner structure inert and still prove properties about it or attach cryptographic signatures.

Collateralized gossip

Vac, a sub-project of the status.im project, has a gossip protocol where you put up collateral in a blockchain and then each epoch you have a quota of messages you can send. This is enforced by a simple construction that may be more versatile than meets the eye. The idea is to prove that you leaked a shamir-secret-share

of a private key that put up collateral but you don't have to say which one (zero-knowledge). The secret share is calculated by using the message as the x-coordinate so if you want to send multiple messages then you will burn your collateral.

This kind of construction may possibly be generalized to deal with bandwidth.

Mixnet

The nym project is a mixnet that uses a blockchain to achieve a client-server protocol in a decentralized network. You can pay to have your data routed through the mixnet and "mixers" can make a "proof of mixing" to earn tokens.

Mixing is the act of delaying messages exponentially (because the derivative of the exponential is the exponential you get uniform randomness to the timing patterns of the traffic) where the messages are encrypted (sphinx packages).

This is theoretically a way to deal with a global passive adversary that can listen to all internet traffic and form inferences about identity based on timing patterns of communication.

Identity

Nym also has some idea about how to do identity, they use *coconut signatures* that allow you to reveal partial information and re-blind the signature so that it always looks different when you reveal the same information often.

However this does not solve the problem of building trust in the certificate authority in the first place. This is where the proof of trust ideas come into play.

The messages you send across trust boundaries are projected from the top level structure (the max-signal lattice) wrapped in network types (as much as possible) and signed with some key. Rather than forcing too much consensus state on the network (and leaking metadata) we'd rather have multiple identities and associate each identity (i.e. public key) with a substructure / category / topic in our max-signal lattice.

Therefore; the outermost layer of the network types must always contain a verification that supplies a confidence in this utterance and a signature proving that the owner of that key was willing to put the stated amount of confidence in the information.

Now others have some preconceived notions about what kind of information this identity is associated with (this could be a totally different vocabulary than was being used by the originator of the message). In this way we allow listeners to chose how they listen and speakers to modify their "tone of voice" (as well as how loudly they speak or rather how *confidently*).

Reproducible Builds

When we have a distributed system that updates itself without a single source of truth then we would like to have strength in numbers. I should be able to compare the results of rebuilding my system with your results and be sure that everyone who is one the same version of the codebase is getting the same version of the binary.

In practice there are two such systems, nix and guix. Both use "deep constructive traces" for tracking whether the build is "clean" and suspending scheduler for ordering the build process.

A constructive trace is a hash of the dependencies of the software to be built. The *depth* of the trace refers to how many layers of transitive dependencies you include in the hash, in practice the two interesting cases are $n = 1$ and $n = \infty$.

With deep traces you capture the whole transitive closure. This allows you to do "shallow builds" where you only need to fetch immediate dependencies and be sure you have a sufficient environment for build (reproducibly!) the system.

With shallow traces you capture only the immediate dependencies so you need to traverse the whole transitive closure to verify that nothing needs to be rebuilt. However, if something needs to be rebuilt at the very bottom, and then the things that depend on it rebuild into identical hashes to before then you can do an "early cut-off" and stop rebuilding dependents downstream.

This is not possible with deep traces, in this situation if a deep dependency changes you will need to rebuild everything downstream (because something in the transitive closure changed).

The elegant way to solve this problem is to do shallow traces for the dependency structure and recursive SNARKs for the proof structure. That way you can get shallow builds and when you do early cut-off you only need to redo the proofs all the way up.

Maybe you can take this even further but attentive readers will see that this is the exact same trick as I am using for zooming around the max-signal lattice and keep up with the flood of information.

Anyway, I consider guix to be the better designed of the two systems. Furthermore, most of the propagator literature is written in scheme (same as guix which is written in guile). The GNU organization is also philosophically very compatible with datalisp; All code is under perpetual review!

The flagship operating system for interacting with the datalisp network will therefore no doubt be guix (more on this below).

Shoulders

It's always good to find good shoulders to stand on.

Hypercore & IPFS

Two enabling technologies are hypercore (which enables the exact idea mentioned in the section preceding this one) and IPFS (which can be used to look up an hash that we come across to see what it refers to, i.e. IPFS is a "bottom storage").

GNU Guix

Reproducible builds are a necessary prerequisite to many of the features in the system (automatic updates via consensus can be built first at one level of trust and switched to after you have confirmation from the network that others are getting the same binary as you).

Pijul

Pijul is a the most general lattice type that we'll need and the core of our system. It holds the code that the build systems then rebuilds and takes note of what changed. Then the relevant datalisp words get new

versions tied to the patch just made.
Everything. Is. Integrated.

Interface

Many of the consensus related tools are reliant on humans and therefore we need interfaces that are idiomatic for managing the datastructure we call "the internet".

Window Manager

Arguably the most important program for managing where you place you attention is the window manager. I can write another whole paper just about this component of the system.

Menu System

The convergent internet index / topology that is being built in the different places of the propnet can be traversed by the datalisp IDE. "The Menu System" is where the "lisp" in "datalisp" comes from. A menu is a list of options and these options are queried for by looking for things relating to a certain data or having certain metadata attached. The menus can be ordered by confidence and can contain executable commands. You can think of them as executable man-pages with a built-in stackoverflow.

The metadata lattice was born out of bikeshedding the ultimate note-taking system. There are types of tops that can be folded into the menu interface, each UI element is a lattice that you can use to orient yourself in the digital space.

Again, I can write another paper about the design of this system (deeply integrated with the window manager).

Tinder

The game of swapping business cards has somehow been associated with different manifestations of these "hookup apps" all owned by the same company: Match Group. These people are making the algorithms that decide how a large part of humanity chooses their mate, finding stable relationships (a relatively basic incentive to assume a user to have) is not aligned with the desires of the companny (active users).

Resolving such contradictions is important but there is another missing aspect which is intent. Users have no way of expressing the intent of their business card, I should have the right to take pictures of my car, put the make and model and suggested price range onto a "tinder card" now someone can click on the pictures to see a description of the car (maybe structured).

The user should be able to swipe on such business cards (looking for flatmates? looking for a flat? looking to rent a chalet with a bunch of strangers that you have high confidence in via transitive trust and go on a snowboarding trip making new friends? want to disrupt global shipping by applying network engineering methodology to optimizing ship speed/size/burn rate but can't afford the multiple trillion dollars investment? how about free hardware? can't afford building your own fab?...). Being able to align incentives and surface interesting information is the key use case of social networking technology.

To just wrap up this thought; public profiles contain minimal information (they are very "sharp", just what is suficient is provided). Each of them contains a public key, when someone swipes on a profile they send one of their own (encrypted) in exchange (as a proposal), this is how key exchange is done.

Profiles naturally form a set-inclusion lattice, since if someone knows that two profiles lead to you they can now unify the statements from each profile into their profile of you.

Guile scheme

Scheme is a very simple programming language, there are only four rules of evaluation (conditionals; condition then branch, functions; arguments then body, values; id, and symbols; values).

It is ideal as a user interface for the moderately proficient user that wants to be able to steal bits and pieces of different configurations and combine them into one.

With the menu system we transform the "search based workflow" that is the shell (and CLIs in general; you type "help" then "help $word" etc.) into an "index based workflow" (basically the next step after searching in google is going through the index of pages suggested). What this means for the novice is that there are always options to chose from.

For example you can ask for commands with the metadata "rsync" and get a bunch of commands with fields that need to be filled in (with data that is specified by some datalisp type... and therefore has predicates) and descriptions of how what will happen if you run the command.

The menu system is made to be an IDE for making such queries and constructing out new paths and connections between different pieces of information (equivalent ways to achieve a task and the different tradeoffs in speed, safety, legibility).

8 Strategy

Let many flowers bloom but let's try to agree on the data interchange format and converge on confluence!

Programming

There's lots and lots and lots of little programs that can be written for the datalisp environment. One of the big reasons for why we want guix to manage packages is that the libraries for the "datalisp language" will be written in all sorts of languages (whichever one you want, we can converge on the legible, safe and performant implementations).

Rust would be ideal for network code and shuffling data back and forth with all this next-gen cryptography. It is a good fit for "politics layer".

Idris has dependent types, is written in scheme (so maybe amenable to seeing guix as *the* idris package manager), and ideal for verifying conformity of datalisp forms and tokens to the declared word. Also for writing transformations that change one datalisp word into another in information preserving or adjoint ways.

APL would probably be good for the bit fiddling (breaking graphs into bitmap and vice versa), it's "natural language independent" and shown by research to be easy to teach to people with some science background (but no programming experience).

Scheme as mentioned before is a good "glue code" and "configuration language" for how we are managing "data interchange".

Scryer-prolog may be a good way to get some of the logic programming facilities of datalisp working. My original inspiration for the programming model was *GameLisp* which exposes runtimes in strategic locations of rust binaries. In the case of datalisp "runtimes" are actually APIs that support some (probably) subset of the available words in the language.

Adoption

A superior form of organization will outcompete existing forms of organization. We should just focus on having fun as we fill in the gaps and wrap existing things in datalisp (for example: you can make toml version 0 for some specific toml parser in some specific language, toml version 1 for another, ... then you can make a version for some toml subset and ways to verify it fits for a bunch of the different underneath words, maybe not all of the toml words end up in the global namespace, nobody cared enough and the most general representation was the one we formed consensus around globally - you see where I am going with this).

Basically; there are no restrictions, this document serves as a general specification of sorts. Feel free to share it and all that.

Education

I was thirteen and just came out of an 18 month severe world of warcraft addiction (and shortly after a binge reading of B.F. Skinners work) when I went to the local university to see some computer guy talk about free software (spoilers: it was richard stallman). I went home afterwards and installed linux, since then my journey has been towards resolving the contradictions of freedom and harmony, towards having security from the nonsense decisions of the world governance.

I believe the most important people to get on board are the kids. My nieces and nephews are addicted to ipads that are mandated by the elementary schools. I don't believe that it is healthy for a child to grow up around proprietary software, you should be allowed to excise magical thinking when you are a child, that is; you should be allowed to follow your curiosity about how things work.

To this end I am thinking about how the guix system can be set up with a session that start you off like computer games where you gradually discover the keybindings for manipulating the system. I have also been thinking about how to build configurable compilers for small languages like forth and scheme so that all you need to do is translate a configuration file to get a programming language in the native language of the child (same for the operating system).

These languages could also be DSLs for programming the menu system, the window manager and the datalisp decision mechanisms. I believe a nice course would take the child through every different programming paradigm over ten to twenty years... If we build an evaluation system that is capable of measuring trust then there is no need for certification, job titles, politicians, judges, courthouses, armies (the largest army in the world is everybody, let's just align our interests).

Longevity

The issue I still see with this system is that it requires you to use a computer and do some sort of "proof of work", I am very sympathetic to aboriginal cultures that are not able to hold onto their beliefs or notions of value due to the entitlement of our economic system.

Maybe a similar criticism can therefore be levied against this system as I make w.r.t. bitcoin at the beginning of this paper.

Efficiency

Maybe the convergence is too slow, maybe the format is too wasteful (why not append the version number onto the operator and save 2-3 bytes per form?), maybe the architecture is too sparse or sends too many messages.

To be honest, it would be crazy if this actually makes any sense, I've basically been bikeshedding what programming language "to learn" for a decade rather than just picking one up and going with it. All I've learned along the way are the constructs that make up the different languages and the tradeoffs they have to make to earn their keywords; "functional", "actor model", "pure", "hygenic", ...

Datalisp is intended to be a data interchange format first and foremost. You should never have to write a length prefix by hand, the language isn't even turing complete (it's just a glorified datalog operating over general lattices with some probabilities thrown in for good measure).

Everything you would want to do when navigating a filesystem is essentially what datalisp is good at doing. So let's forget about filesystems, enter the age of nonvolatile memory and content addressing in style; with csexps and logic programming.

Security

Security always has to be considered in relation to your threat model, however, irregardless of threat models there are certain properties that help with security. Such as, legibility and having a clear idea of where intrusions can come from. When you base your whole computing experience on binary blobs, millions of lines of C code or undocumented hardware, then your ability to secure yourself is severely compromised.

Any project that wants to be a foundation for a digital society, that is; a p2p network, must take the question of legibility and *learnability* very seriously.

Datalisp is built on canonical S-expressions because they are the simplest context free grammar (which is sufficient for describing a turing complete languages, therefore anything) and every piece of data will have an explicitly declared type that is forever backwards compatible and therefore increasingly documented with time. It does this by having every word in the language versioned with a natural number and using a byzantine fault tolerant consensus algorithm for coming to global consensus on what every word means.

Furthermore, and perhaps most importantly, you can exert full control over your data and there is no one that can access it unless you opt into it.

By having the words hide arbitrarily complex constraints and using probabilities to reason about legitimacy we can hide a lot of structure inside of well defined binary formats (which csexps support).

This way we can give people the ability to inspect their data (even writing their own tools to do so) and due to canonicality we can reduce the amount of duplication of equivalent data, meaning that in a content addressable network there will be more seeders for any piece of data than there otherwise would be. This allows you to hide in the crowd and gain greater confidence that this data has many people depend on it.