Datalisp Technical Report

ilmu@rishi.is

ABSTRACT

This paper presents some of the potential of canonical S-expressions as a data interchange format by showing how they can be used as a basis for designing a system with many desirable properties. The presentation walks through; a description of design considerations, (some of) the tools needed to interact with the data, how the system can be used as a metaprogramming framework (i.e. operating system interface) and finally we sketch a coordination mechanism for a decentralized name system and end with a discussion of how one could potentially create a sustainable decentralized network.

CCS CONCEPTS

• **General and reference → Computing standards, RFCs and guidelines**.

KEYWORDS

datalisp, data-interchange, P2P, canonical S-expressions

ACM Reference Format:
ilmu@rishi.is. 2022. Datalisp Technical Report. In *Proceedings of European Lisp Symposium '22 (ELS, 2022).* ACM, New York, NY, USA, 7 pages. https://doi.org/idkwhattofillinhere

A reoccurring problem is picking an optimal tradeoff between efficiency and redundancy, latency and throughput, consistency and availability, censorship and pollution. This can be characterized as different manifestations of Ohm's law, in many ways you could say the goal of engineering is to navigate this kind of tradeoff space.

In software engineering we come across this tradeoff quite often, one manifestation is known as the "lisp curse" - that is - lack of censorship causing pollution in the language because everyone extends it in different (perhaps incompatible) ways.

1 INTRODUCING DATALISP

A *Motzkin path* is a word with digits from the alphabet $\Sigma = \{()\_\}$ subject to the constraint that parenthesis must be balanced. *Canonical S-expressions* (csexps) are Motzkin paths where the underscore is written as a `netstring`. A *netstring* is a length prefixed string: `k:<k-bytes of data>` where `k` is an ascii encoded decimal. Syntactically correct canonical S-expressions are in one-to-one correspondence to the datatype Tree [u8].

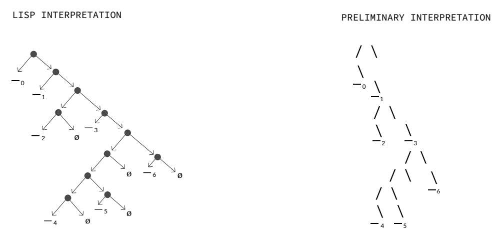
At this point it is good to pause and take another perspective. As a data-interchange format the above specification is complete, however, not all syntactically correct canonical S-expressions are simple to decipher for the receiver of such data. What semantics are implied by our syntax?

ELS, 2022, March 21–22, 2022, Porto, PT
© 2022
ACM ISBN idk what to fill in here…$15.00
https://doi.org/idkwhattofillinhere

There is *quotation*, i.e. length prefixed data, and *composition*, i.e. parentheses. If we could *name* compositions then we would have a very economical way of moving around snippets of data and composing them into different programs. This motivates the need for a name system, which means we must find some sort of data interchange *mechanism* to drive it (i.e. to give the names meaning).

One thing to consider for a data interchange format is that it deals in "facts", everything is "self-evident" there is no evaluation where you rewrite this into that in some globally comprehensible way (since arbitrary programs can be written to read and write our data format). When we consider the semantics with which lisp reads S-expressions then we can see that it's a structure meant to be permuted heavily (lots and lots of pointers). However, datalisp derives its name from datalog and datafun so the implied semantics are different, for now you can treat a form like `(f a b)` as a mix between a clause `f :- a, b.` like in datalog and an f-expression like in kernel [1] and other such lisps.

$$(\_{}_0 \ \_{}_1 \ (\_{}_2) \ \_{}_3 \ (((\_{}_4) \ \_{}_5)) \ \_{}_6)$$



Given our current understanding; the zeroth underscore is the *name* of this data and for that to hold true the rest of the form must conform to the specification of that name. However we don't know the limits of this "specification" yet so we can't say we know about all syntactic restrictions (that we care about) yet. For example we currently aren't able to say anything about the way that the second and fourth underscore are interpreted due to the excessive power of the zeroth underscore (later we will see that it is unlikely that data with this shape is complete).

It would be good to do a sanity check w.r.t. our design criteria (lisp curse). The way we do this is to consider equality, we want equivalent data to be equal data. This is often achieved by running a domain specific canonicalizer (compiler) on the format in question. While canonical S-expressions are already canonical w.r.t `Tree [u8]`, we would like to do better: our data interchange mechanism should enable coordination of canonical representation (allow peers to discover shared semantics) and if possible encourage it! However, since we don't have a stick to wield (p2p), this "encouragement" must be via carrot; which is, in some sense, our ultimate goal: converging on confluence via economic incentives.

2 TEMPLATES

In order to achieve this goal we first need a way to refine representation of semantics. We achieve this with templates, this is the first part to understanding how we model the system from the "outside" (i.e. from the perspective of the data rather than the programs - "show me you tables and I don't need your flowcharts").

One of the most fundamental tools for interacting with our datastructure is a `less`-like program (a unix `$READER`) that allows the user to highlight text and persist the highlighting. When the user exits the program it will return the data needed to recover the session. We represent this data as two lists, the former one quotes all the non-highlighted parts and the latter one quotes all the highlighted parts. To recover the file we interleave the two lists (unquote and concatenate) and highlight the text from the latter list.

```
This sentence is false.
     -----------

Frame: (1:25:This 7: false.)
Vars:  (1:111:sentence is)
```

Since we want all data to be named but these lists have no inherent meaning beyond how many elements are in them we've found the first useful names: the natural numbers as ascii encoded decimals (length prefixed lists essentially). These are reserved names in datalisp. All "anonymous data" is named in this way unless there is no composition. This will become clearer as we discuss string diagrams more, suffice to say that the simplest diagram is `n` side-by-side identity wires (i.e. `n` quoted data).

We call the former list a *frame* and we consider it "immutable" in the sense that we will hash it and expect to be able to fetch it again by content address. The idea here is to be able to link a lot of data to the frame (such as: locally available data or default values for filling the holes, predicates for testing data before putting it in a hole, programs who will accept the file resulting from filling the holes w.r.t. predicates, annotations, explanations, etc.) and then we *name* the association of this linked data with the frame. We'll refer to these elaborated frames as *templates*. The name allows us to easily refer to templates when we want to pass data into the system safely. Naming templates in a "metastable way" - that is; when the template receives bug fixes we want it to be likely that others will believe that the new data is what the name refers to.. but if someone claims the name refers to malware then we want it to be unlikely that anyone will believe the name has that definition - this is a complicated problem that we will scope out a bit in a later section where we discuss legitimacy.

For binary data we need to think about byte alignment. This means that for each quotation of binary data we need to read the first three bits to know how many bits after that we should ignore before we start reading. If the binary data happens to be byte aligned then we can skip this, we've motivated a small syntactic addition to our canonical S-expressions: `k;<k bytes of data>` where the semicolon means that the data is not byte aligned binary data (and therefore will use this three bit trick), whereas a normal colon means our data is byte aligned. We will name these kinds of netstrings differently; "nitstrings" (b**i**t vs byt**e**) because I will nitpick that this small addition would undermine the canonicality of our encoding so I am against using it without proper forethought and I prefer moving such bookkeeping into the name (in practice it is easy to specify which slots will have binary data in the same way as you would specify any other property).

A so far implicit notion is the idea that canonical representation is so important that we are willing to use an inefficient format (on disk representation, network packets, etc. would not use canonical S-expressions). Hopefully this paper serves as some motivation for this belief, content addressing is already a strong argument but clarity is also important.

What we've described so far allows us to abstract interfaces to arbitrary programs and gives us further understanding of what we actually need, semantics wise. For example we know now that each name can be "flat" in the sense that it doesn't change the meaning of names in sub-forms, the goal is only to collect all the quotes needed to plug the holes in the frame of the template associated with the name and produce a complete file (which will be understandable to things exterior to the system).

Here we run into another tricky problem; method dispatch. For the proof of concept we will use a simple approach; we associate names with frames without ambiguity. That means each name has a fixed number of slots because each frame has a fixed number of holes. However we will often have default values available and having many names for different states of partially filled in data is cumbersome, likewise we want the name to refer to a template not a frame and there are many templates for any given frame. In order to have a "new name" for these cases we will adopt the convention of versioning the templates for a given frame.

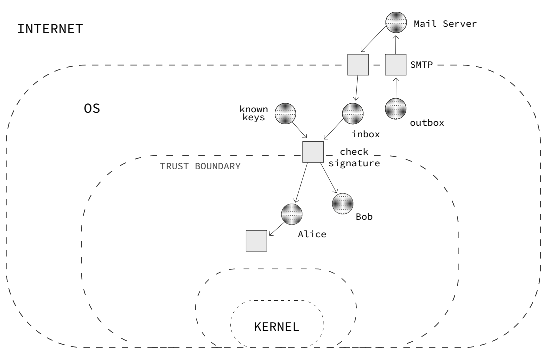
This means we have `(definition name k specification)` where `k` is the version (as ascii decimal) of the name and we will refer to `name` by version so for example `(4:name1:0...)` is the zeroth version of `name` and refers to some unique specification. At this point it is becoming clear that a specification is two things; the templates for invoking programs to check data and guix packages for reproducing those programs. Together they are the information required to check whether a name is being used correctly.

Composing templates by putting templates in holes of other templates results in a tree of templates whose holes are the holes of the templates along the fringe of the tree (as well as unfilled holes in the interior), we would want to identify holes in some cases and shrink the interface. This kind of problem is described by string diagrams which we will be assuming some familiarity with from here on out, a gentle introduction can be found at [2].

With logic programming we have a way to ask for paths between names and put rules on compositions. This allows the user to access different representations of data seamlessly and a canonical representation also makes the programmers job easier (implementation complexity is $O(n)$ via star topology).

3 BIPARTITE GRAPHS

In lisp we try to exploit the duality between code and data to simplify the task of programming. However we rarely represent the duality with a bipartite graph… Consider the following sketch:



There are potentially many ways to understand this sketch but in this section we are more concerned with the structure than the content. We dissect it (roughly) as follows:

- Squares are called “transitions” they are represented by a guix manifest (describing how to reproduce the programs needed to validate the data and perform the transition). We call this the meet in the transition. Transitions also describe (using templates) how to wire up incoming quoted data to call the programs correctly; the join in the transition.
- Circles are called “places” they are where we know what names mean; each name is specified and the associated data must be valid according to that specification (guix packages + datalisp templates). The same name can mean different things in different places. The meet in the place is the collection of specifications and associated names (i.e. the local definitions) while the join in the place is the cached data meeting those specifications (you can think of the join of a name in a place like a table in a database).
- Circle->Square arrows are the names [from the source place] that are allowed to fire this transition and how to wire them to the transition interface.
- Square->Circle arrows are also compatibility parts the data here is projections, injections and rewiring of that kind between (some of) the wires from the transition interface and (some of) the named wires in the target place (that place will test the received data according to local customs).

The arrow data is there to cover place-specific compatibility with the transition interface, The idea is to give each peer some flexibility so that it is easier to have independent consensus on the transitions (likewise for places).

The reason we speak of a "meet" will be elaborated below but the idea is that we want the peers in the network to be able to agree on transition representatives and in that way they pool their risk. These kinds of "eggs in one basket" gambles can then be used to measure the legitimacy of branches in the source tree ("how bad would it be if this transition is faulty?" == "how many people depend on this?" × "how much can damage can it do?", this is a similar idea to "bus factor" except here we don’t care who maintains code; we care who depends on code). The idea is that if everyone is willing to trust something (newtonian physics..) then it is likely to be legitimate but there is also a greater reward for proving it is not.

Pooling risk makes it economical to improve the security of the whole (cheap for each person to contribute but still in aggregate there is enough to pay for proper infrastructure maintenance) but it also makes it harder to trust anyone with the associated responsibility. We will however (again!) postpone discussion of this problem and continue with our reification of the sketch presented.

There are 6 slots under *bigraph*:

- A first slot is raw binary data encoding a $n \times m$ bit-matrix for $P \longrightarrow T$ so the bit at index $k \cdot n + r$ indicates the existence of an arrow from place r to transition k (if I got the orientations right).
- Similarly, we encode $T \longrightarrow P$ as a $m \times n$ bit-matrix.
- Then we have four lists, the first two: are the places and transitions (of length m and n respectively) they describe our database of stored data in the case of places and programs ("stored procedures" in databse lingo) in the case of transitions.
- The latter two lists are the arrows: first of length equal to number of 1 bits in $P \longrightarrow T$ and latter has length equal to 1 bits in $T \longrightarrow P$. They have the data representing what we discussed above.

Of course we take advantage of content addressing to chunk this up into manageable amounts of data but in principle this bigraph is the "proof" that we want the computer to produce. The goal is to find a way to incentivice improving the proof so that more peers can trust it.

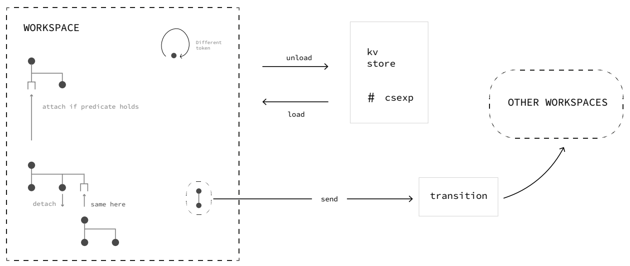
Anyway, let us return to the matter at hand by expanding our understanding of the aforementioned "outside view" - if you consider how a user treats data they are processing in the shell; caching it in files before making experimental changes that eventually work as intended - then clearly this workflow is improved by forgetting about the nitty-gritty of files and rather just quoting all data at rest, that way we can work on many pipelines in parallel and capture the resulting dataflow construction as a bigraph that others can reproduce and make use of.

There is no reason we can’t have an interactive GUI to access places and fire transitions, in fact statebox [3] has already implemented interactive graphical editors for string diagrams and petri nets. Furthermore; you’d need to be psychotic to program csexps by hand, therefore, we should associate templates with other informative presentation formats [than bigraphs and string diagrams] this motivates general purpose structured editors for csexps (with different layouts depending on the data in question) and a "menu system" for firing transitions and traversing the database (that I will describe in more detail below).

4 VERSION CONTROL

We've already talked about how the transitions are like build systems since they are concerned with reproducing the correct code (given some data). What we've yet to talk about is how the places function like version control systems (since they care about data).

A recent paper[4] shows a way to do linear time diffs on typed trees (or ordered trees). Canonical S-expressions can therefore be efficiently version controlled (by treating the netstrings as immutable we find the minimal conflicting substructure in the tree and replace it).



This picture is based on the ideas in the paper but I am sure we can go further by integrating ideas from pijul and using some classic algorithms to change tokens into frames (again trying to find maximal shared structure, this time in a flat `[u8]` file) and then using the structural methods described above.

Sadly I have not yet reached very deep in this investigation so I will not go into the specifics of "how" and rather will focus on the "what"s that have nice implications.

We can think of each place as a workspace like in the picture (backed by a datastore to retrieve preimage of hashes), it is constantly looking for shared structure and keeping track of the content addresses it has seen. This makes it easier to reason about idempotence (which can grant us coordination-freeness if we satisfy propnet axioms) and allows us to version interfaces, since we can find which edges have to be added to the graph so that we can "lift" a place up a version and then eventually if we trust the new version enough (i.e. the diagram proved commutative) then we can "drop" the old place from the bigraph (but keep the arrows needed to route past it).

Any invariants that should hold such as adjunctions or commutativity of (sub)diagrams in the bigraph can be tested for programmatically as new data arrives in that part of the graph. This way we can build confidence in new versions. Any broken invariants that we detect we should share with the network to obtain economic gains of some sort (i.e. the protocol rewards sharing useful information such as counterexamples to theorems).

Type theory is useful for reasoning about composition and in datalisp we want to support different user provided type theories (to aid in program construction and other metaprogramming). The logic programming that is responsible for maintaining invariants in our system is not fully worked out yet but it is also out of scope for this paper; the general idea we are pitching is to replace a file system with a some kind of general-purpose type system.. but investigations into the intersection of logic and measure theory (and especially how to implement these theories in computers) is a very deep field (called artificial intelligence) and from what I can tell it offers diminishing returns to dive too deep into this too early. I think it will be sufficient to use an algorithm like datalog (query transitive closures of some rules+facts) as the engine of the proof of concept implementation, that can then serve as reference for further development.

As mentioned above, the form `(4:name1:0...)` is associated with a specification indexed by version `0` of `name`. A very type system-y idea is that if the graph updates arrive via the graph then we should be able to always keep the version<->implementation relation coordinated with peers we trust (we model them as places in our graph as shown before in sketch so we just need to add a transition that normalizes any quirks in their vocabulary), to achieve this we will want to keep track of where information comes from and what vocabulary different peers are using, then use structural decomposition to remove confusion and keep track of causality (see all papers by Ugo Montanari, he is incredible).[5] To improve this further we'd want to be able to negotiate with our peers to standardise or simplify things.

Among things in need of simplification is equivalency classes of places and/or transitions (they will usually be equivalency classes since whenever any dependency is updated there is one more possible *meaning* for the name).

We've mentioned several times already how the goal of datalisp is to be a (sustainable) decentralized name system, i.e. to resolve the ambiguity caused by multiple equivalent things and allow users to refer to semantics by name, qualifier free, without unexpected things happening (i.e. "do what I mean"). Now it is time to start talking about how to measure different candidates and how to coordinate with other peers in the selection process.

We will need some measure theory and I again am not expert enough to magic one forth. However I have a pretty reasonable approach to testing different candidate solutions to the problem. The general idea is to use a protocol where you trade `(signed (contextual (measured (named data))))`, there will be a more thorough look at this when we describe a potential bootstrap version (with each field in that expression explained and a proper introduction to the problem it tries to solve) but first we need to talk about UX.

Since the system is peer-to-peer there is no root of trust, therefore, we only have a measure of trust for each exterior information source (in each context) and perhaps a measure of our confidence in that measure. We want to make this information resistant to manipulation by cooperating with people we trust in reality, this is how we obtain a (so far informal) notion of "proof of trust", how we motivate people to share their truthful estimations and whether or not these estimations can be made to converge in some manner is a question for religious study, let's return to tool making.

5 INTERACTION

The menu system has the guiding philosophy that it only provides you with a window into data where you can perhaps perform some structural manipulations via keybindings but then those changes are atomic so that it is always possible to serialize the current state and share it with others or reproduce it yourself.

We've already discussed how we can create/view/edit frames and while that is an important piece of the system it should be clear by now that the higher level abstraction; templates, are going to be important. Now is the time to talk about the associated interfaces.

- *Editor* - the simplest structured editor, similar to what you'd use to interact with a filesystem, i.e. move, delete, edit and add vertices in/to/from a tree (like files and folders in a filesystem). This editor is used to make syntactically correct changes to csexps, it also (optionally) outputs the shortest editscript for the session (i.e. the data of what changes you made, which is allows you to undo them / share them).
- *Menu* - while this program seems simple; "just a way to fire transitions in the bigraph", it is actually an internet browser. The evolution tree for this program is huge but the simplest manifestation is just a list of human readable options backed by shell commands. Select an option to run a command. By adding a minibuffer we can show the command to run or annotations or any other (supported) metadata about the highlighted option. Options can of course open the menu program again but with different data. This makes it easy to think of various utility programs that can hook onto the menu as well as different menu layouts (underneath it is all just data). In particular it would be useful to have interfaces to fill in templates, that way the menus can index partial commands (with constraints) which means each command can be more general (easier to coordinate) while still being safe via predicates (and legitimacy contest).
- *Shell* - Combines the two menus above with an interactive "shell language" which is a human-typeable syntax that translates back to canonical S-expressions. This time our number one design criteria is tab completion but we would also want a clean way to describe firing sequences in the bigraph.

The structured editor will eventually need to support predicate checking data in the tree and external viewers / editors (for images/videos/etc) but otherwise will most probably be practically replaced by more domain specific utilities as the ecosystem matures (however in many ways it is the ultimate fallback since it is general purpose).

I'm pretty sure that all these interfaces will converge in the shell language IDE program. The shell syntax is based on "generalized S-expressions" which are still work in progress but are meant to give easy access (via our vocabulary of names) to all the code and data that we know about in this computer. For this syntax to be useful we require that any syntax rule must have the editor support to unambiguously rewrite back to a canonical S-expression in a user-understandable way.

In my current design the interface is split into four parts:

- *header* is one line at the top of the screen indicating stateful things like the mode the editor is in and the location in the contexts hierarchy.
- *viewer* is the top half of the window, it will elaborate what the highlighted slot contains (rough intuition: it follows one pointer out and displays what it finds). At rest it will show the bigraph we are working in.
- *options* is the bottom half of the window, here we see the possible things to hook in at the current location (if we are tab completing) or possible structures to explore in the viewer (links out from currently viewed data or similar).
- *commandline* is one line at the bottom of the screen where the user types in generalized S-expressions.

Okay, so what are these gsexps? We already mentioned how tab completion is an important consideration (as well as human type-ability..) so it may not be surprising that we are whitespace sensitive and have the equivalency: `(f a ...)` is `f(a ...)` is `a.f(... )`.

However we don't really care in which order the template is filled so we can add a bit more syntactic sugar `(f a b c)` is `(f' b c a)` is `(f'' c a b)` this gives more accessibility when tab completing, ideally the application has enough polish to auto rewrite `f'''` into `f` and keep the user in flow. Similarly we need a way to "box" whitespaces in a quote, the shell will therefore be using a vim-y command language for entering and exiting boxes (quotes, netstrings, they are all the same).

Finally we also have three types of commonly used delimiters: `(parens)`, `[brackets]` and `{braces}`. One idea is to use parens to access names in places, brackets to access names in transitions and braces to access discrete data via different search methods (higher order filters). For example if you finish filling in a parenthesis and send it to the bigraph then a token is put in that place on the bigraph. If you fill in brackets then you are asking to fire a transition which could also put some tokens on the bigraph. You can then serialize the marking on the bigraph to save what you were working on and return to it later.

The main takeaway is that we can use different parenthesis as different access patterns to the data in the bigraph and in that way make the tab completion more "complete".

Anyone who has played with the analysis tool in lichess or used agda will know that it is possible to provide very good UX for structured editors, the key problem is how we organize the data and therefore the root problem is finding a method that will converge on a solution. If the convergence is bound to happen then our problem is said to be "coordination free" but resource allocation problems always require coordination and cognitive capacity ("namespace") is a scarce resource.

Trust is the thing we want to quantify, we define it as the thing that turns information into behaviour, so what we want to assess is which information (data) we want to turn into behaviour (code). These assessments are useful for others so, as previously mentioned, we'd want to encourage peers to share their truthful assessments.

6 COORDINATION

In a peer-to-peer system we have no a priori hierarchy but we may have some common starting point (synchronization); software that all honest peers share. We call this the *meet* in the network our goal is to give the honest peers a way to trust strangers without being burned by byzantine actors.

Sustainability is an important property for a decentralized system, it is defined in terms of the incentives in the system; a *sustainable decentralized system* is resistant to *the tragedy of the commons.*

The tragedy of the commons is a famous problem in economics, defined as "the interest of the individual colliding with that of the whole" so again the "lisp curse". Another way to state it is as a non-convex incentive structure, that is, where locally optimal behaviour can undermine the functioning of the system. This problem is usually solved (in theory) with centralization (censorship). However that is a bit like postulating a solution since the original problem is precisely that we don't know how to align incentives (i.e. centralize in a "fair" manner), systems like taxes and voting are still vulnerable to the tragedy so government is not really accountable. This is a huge problem in practice.

The nobel prize economist Elinor Ostrom stated (famously) that "a system that works in practice can work in theory" (about sustainable decentralization observed around the world) but we don't know if that is necessarily true (since we cannot rule out the existence of god $\in \emptyset$). What we can do is build a realistic representative (a "personal bureaucrat") that actually has the users best interest in mind and then hope for the best, maybe it works in practice. On that note: I think any realistic attempt at peer-to-peer networks should be mostly focused on making the peer more representative of what the user wants. To that end communication protocols are motivated by the desire to be precise in your communication (for fear of being misunderstood) while still being able to shitpost occasionally (like the hackernews slang of ending a message with `/s`).

We want to use a similar strategy as the quecha (natural) language, where metadata (grammar) attached to messages gives indications about the source of the message and how much we trust what we are saying. As indicated before our blueprint is `(signed (contextual (measured (named data))))` where the proof of concept will interpret the blueprint in the following manner:

- `data` is an IPFS hash that links to "anonymous data" so if we have `(x:namey:somez:data)` then we will pack it up as `(x:namew:ipfshash)` because the metadata just needs to tell you how to retrieve the data in question (which you will do if you think it is worth it) so the data we put into IPFS is `(1:2y:somez:data)`.
- `named` is the name we believe should be associated with the data. If you want to make second order statements about your beliefs (like "I believe that this name is what peers X will use but that name' is what peers Y will use") then the `named` is a larger structure than literally **just** a name (and maybe a version) but it's better to build vocabulary bottom up and be concise with messages.
- `measured` means that the signee has voted with an assessment (which is implicitly a bet) the ballots for the proof of concept network will just give the odds of Signal vs Noise (information you will trust vs you won't). In his book[6] Donald G. Saari observes that the Arrows impossibility theorems are simple information theoretic results that are caused by the ballots not giving enough information to produce a reliable tally.
- `contextual` refers to the context that measure contributes to, essentially it is a way to tally the votes. The context that you are participating in coordinating will have some rules for collecting up the tallies, something that isn't really worked out but I will share my intuition.

We've already mentioned how coordination-freeness is a property of a converging system (where it doesn't matter in which order information propagates around the system) and how resource allocation is not a coordination-free problem, in fact I believe resource allocation to be a coordination-complete problem (every kinda of coordination problem is a resource allocation problem and vice versa).

In the system we have two types of *consensus* (which is *coordination* that we have blessed as *legitimate* rather than *accidental*), *meet* consensus is formed by many peers intersecting in some common functionality (like the base OS and "automatic updates") while *join* consensus is a peer consenting to some configuration (like installing software).

- Meet consensus; the peers have places seeding the agreed upon data. Each participant in the consensus is lending legitimacy to this version of the software in question, this makes it more likely that new users install the legitimate version. All further development is likely to build on the most legitimate meet so a meet consensus can be seen as a "release" of the software.
- Join consensus; when a peer decides to add customization's or patches from the internet to their graph. It's essentially the same as installing software, of course many peers in the same join can form their own meet (release of the software) that "leaks" legitimacy to their dependencies (or seen another way; derives legitimacy from associating with trustworthy dependencies).

Note that each peer is only able to make probabilistic estimates about the outside world so they may not actually form global consensus (or any consensus for that matter, maybe you are only communicating with bots and you'd never know). These estimates are calculated by updating prior assessments of incoming messages, multiply the foreign prior (odds of signal) with our local assessment of the trustworthiness of the source (a bayes factor) to obtain a posterior. We can now sort data by priority based on this posterior, any feedback we receive from the user updates our local assessments rather than getting leaked to the network.

Sharing our local priors to the network is done (indirectly) by sending in a tally. The general idea is that each peer calibrates the voting rights they give to others as the discussion unfolds and then collect the votes (weighted by legitimacy) into a tally that they share with the network.

Any two peers always have an implicit bottom of *Void* (i.e. they have never communicated, even indirectly, no software in common) and an implicit top of *Conflict* (i.e. they can't agree on how to proceed with their communication, to the point of death). Currently I am thinking about a system where peers can establish a bottom (a context) via initial meet consensus.

Participating peers put polynomial commitments in the bottom along with (zero knowledge) proofs that the degree of the polynomial equals some n and that the constant term of the polynomial is the private key of the participant (i.e. fits with the attested public key). The idea is that the participant can now send messages as a representative of the meet (collective identity) but they must prove that they are leaking a point on the polynomial when they do it. Their influence on the context is limited by the degree of the polynomial (once an identity is burned it becomes "anonymous" so we lose any predictive power). In order to replenish their supply of influence they must continue to coordinate the collective identity.

This is essentially a twist on the waku protocol which is a twist on shamir secret sharing. Our goal is to make a self-regulating monetary system based on having this kind of artificial(?) scarcity in ability to communicate. Each peer will constantly have to be establishing new contexts by making alliances in the fight for legitimacy (so that it can sell-out or take advantage of the influence it has on other peers, hopefully at the cost of its legitimacy [i.e. continued ability to do noisy things] if so $\Rightarrow$ byzantine fault tolerance).

What the *context* has to be able to do, is to resolve an epoch of communication (w.r.t. a meet) by comparing all the tallies and seeing if there is enough consensus to continue or if there is conflict. This will result in a proof that is in the meet of the next epoch.

Given some peers representing a meet consensus that many have joined (i.e. it is very legitimate), if their interest is aligned then they all work together towards a common goal. If however, their interests are conflicting somehow then maybe they cannot make progress with their development; causing stagnation. This may not be serious in the short or medium term, but eventually wayland shows up (i.e. some alternative). These other choices put pressure to cooperate on the participants (in order not to be left behind by the competition) or move to greener pastures (coordinate something else). Forks can also gain legitimacy in such situations.

Earlier I called coordination "taking a consistent quotient" so I should define *Quotients* are created by identifying equivalent data and choosing a canonical representative for the equivalency class. This can for example be a way to manage multiple drafts of a paragraph in a document or a huge comment thread.

Metadata describing how data is identified or distinguished in different contexts can be looked up by content-address (i.e. asking the network for data referencing the hash), the idea is to use (probabilistic) logic programming to drive the data interchange graph and order incoming messages to protect the users focus from noise but we leave further investigation for a later date.

7 CONCLUSION

I hope we can make canonical S-expressions into a legitimate choice for data interchange. Some basic tools for working with them will go a long way but in order for them to become the canonical package description format or a programming language agnostic type system or even a digital democracy system/cryptocurrency - then it will require some coordination.

However that coordination is long overdue! We desperately need a way to refine legacy interfaces (such as dotfiles of various formats or APIs that are too large to reimplement bug-for-bug coughXcough).

Having canonical representations for even "simple" things, like keybindings, makes it easier to compile configurations for different choices of window manager / text editor / etc. In a way that is consistent across the whole system. This then makes it (much) easier for a new software to know where they "fit in" i.e. what kind of interface do they need to support to be painlessly compatible with the interface people actually use etc.

Similarly new internet protocols are completely inaccessible unless blessed by google to be supported by the one true browser. By having a simple method to index shell commands (with their dependencies pinned via guix) it becomes much easier to accidentally share knowledge and your interface becomes more protocol agnostic.

Where things get more complicated is keeping track of propagation and managing the probabilities. This is probably going to get arbitrarily complicated once market incentives are in the mix and I'm pretty sure solving that problem would be no less impressive than solving AGI.

Although solving it would be hard, we can get a long way with heuristics. In a content address based network, canonical S-expressions look to be as good a solution as we can reasonably come up with and bigraphs with topological semantics seem like a pretty safe place to start. Although datalisp (as specified) does not need to use the parenthesis we must consider byzantine fault tolerance ;)

Finally, I believe that there is enormous benefit from having a "type system" like datalisp for managing compositions of programs. We want it to be easy to isolate some behaviour and gradually rewrite software to be more trustworthy. An emerging standard for data interchange (as opposed to an imposed one) would perhaps allow us to balance censorship and pollution and overcome the lisp curse.

REFERENCES

[1] J. N. Shutt, *Fexprs as the basis of Lisp function application or $ vau: the ultimate abstraction.* PhD thesis, Worcester Polytechnic Institute, 2010.
[2] "graphical linear algebra." https://www.graphicallinealgebra.net.
[3] "Statebox." https://statebox.org.
[4] S. Erdweg, T. Szabó, and A. Pacak, "Concise, type-safe, and efficient structural diffing," in *Proceedings of the 42nd ACM SIGPLAN,* PLDI 2021, (New York, NY, USA), p. 406–419, Association for Computing Machinery, 2021.
[5] U. Montanari, H. Melgratti, and R. Bruni, "Concurrency and probability: Removing confusion, compositionally," *Logical Methods in Computer Science*, vol. 15, 2019.
[6] D. G. Saari, *Basic geometry of voting*, vol. 12. Springer Science & Business Media, 1995.