Datalisp Technical Report

A reoccuring problem is picking an optimal tradeoff between efficiency and redundancy, latency and bandwidth, consistency and availability, censorship and pollution.

This has been referred to as “the lisp curse”; lack of censorship causing pollution in the language (since everyone extends it in different ways).

I don’t claim to have a closed form solution but I would like to make the problem easier to manage… What started off as a data-interchange format has now become a metaprogramming framework (in my mind at least).

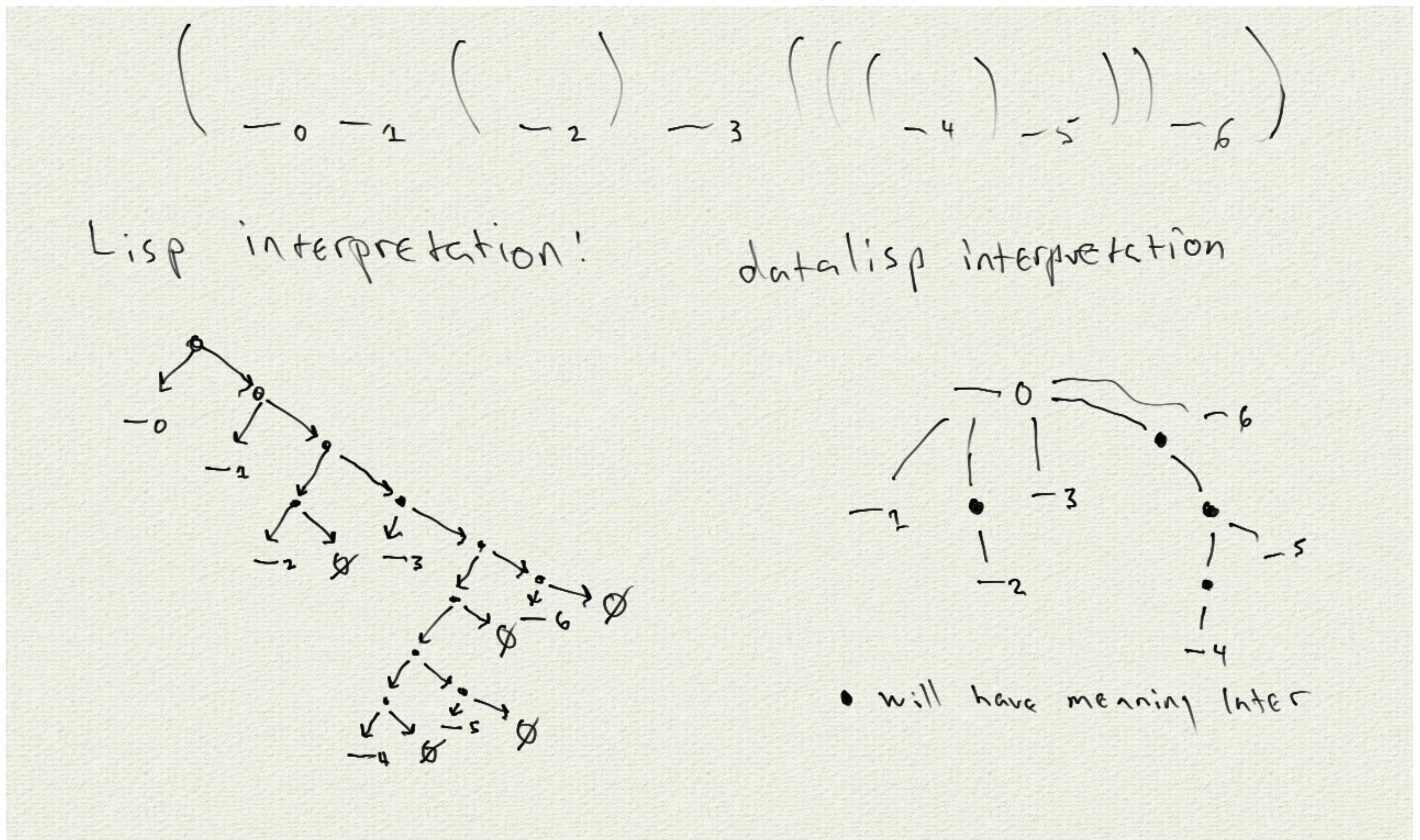
Introducing Datalisp

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.

In lisp we read certain semantics into S-expressions, namely that of a semifull binary tree (cons cells). However, datalisp derives its name from datalog and datafun so the implied semantics are that of an implication graph or propagator network.

So we have the datalog interpretation: `(x y z w)` is `x <= y ^ z ^ w`.



Typed Templates

Consider a type signature in haskell; `f :: a -> b -> c` , this notation makes sense due to currying and lazy evaluation. However in lisp we'd have to describe things like that ourselves, the default idea is to treat the operators like tags for data resulting from evaluated form; `(f a b) ;; returns c` is more idiomatic as `(f->c a b)` or some contextually appropriate way to indicate `c` to the programmer via the name of the operator.

With datalisp the idea is to use templates to abstract interfaces to arbitrary programs. This allows us to refine representations of semantics.

The data pictured above would be considered partial (the dots indicate missing context ~ ambiguity given our semantics). The zeroth underscore is associated with a predicated template and then all the other data has to fit into the slots in the template, by giving partial data you are creating constraints to help you search for the missing data. More on that below.

A template is simple to create and describe, as of now; nearly nothing is implemented but work has started at git.sr.ht/~ilmu/sbcl-tala (http://git.sr.ht/~ilmu/sbcl-tala), nevertheless, here is how it works:

Given a text file open it in a viewer similar to `less` or `vim` - the user will have the capability to highlight text and persist the highlighting. When exiting the program the stdout will have a canonical S-expression consisting of two lists:

- frame: `(13:a good story 12: never ends 6: soon.)`
- variables: `(45:(such as the hitchhikers guide to the galaxy)22:unless you read it too)`

To get back the original file you can interleave the two lists and highlight entries from the second one:

> a good story **(such as the hitchhikers guide to the galaxy)** never ends **unless you read it too** soon.

Once you have punched holes in some file and created a frame, then you can attach predicates to the holes in the frame. Then once you fill in the holes you have (supposedly) a correctly formated file that can be given to the program associated with the template.

A template name is therefore associated not only with the frame but also with the predicates that check the holes in the frame and with the program that will evaluate the filled in template.

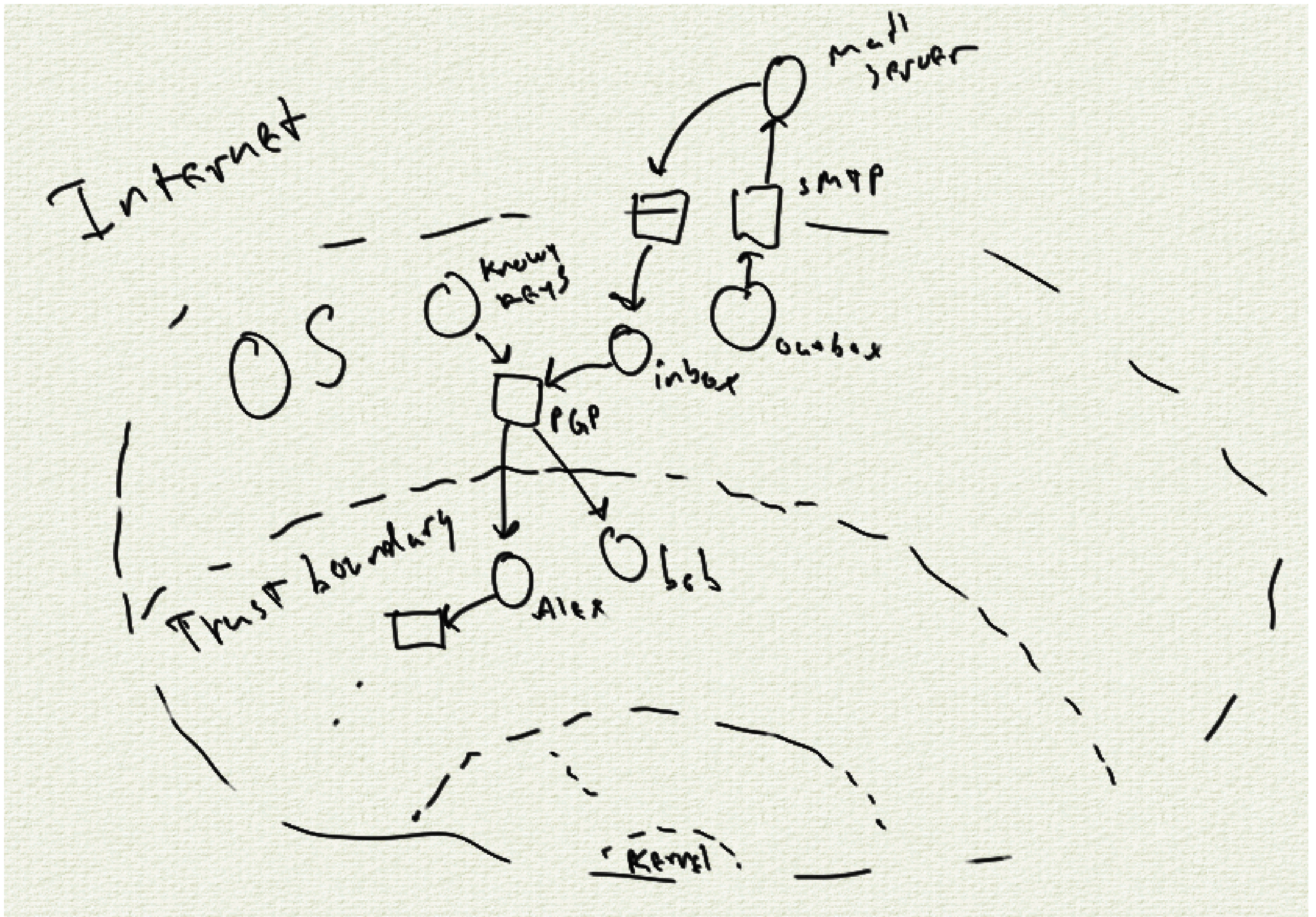
What if we want to put a template in a hole? Well then we must first fill in that template and then test whether the result fits in the hole above.

As a result we can have a joinsemilattice of templates where we have to fill in the fringe before proceeding up the structure; collapsing full templates into slots along the way till we've built the resulting file.

Making sure that the predicates cover the holes sufficiently well makes such builds more likely to succeed (backtracking could potentially be very expensive).

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 we disect it (roughly, I haven't found the right way to factor this so it's wip…) 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).
- Circles are called "places" they are represented by a namespace, each name is guaranteed to only have valid data attached (stipulated by the local definition).
- Circle->Square arrows are descriptions of how to call primitive-predicates and (logic) compositions of predicates to check holes in templates that are named in the place (source vertex of arrow).
- Square->Circle arrows are descriptions of how to call procedures, they may have templates for arriving data (before invoking procedure) or post-processing to fit into destination names.

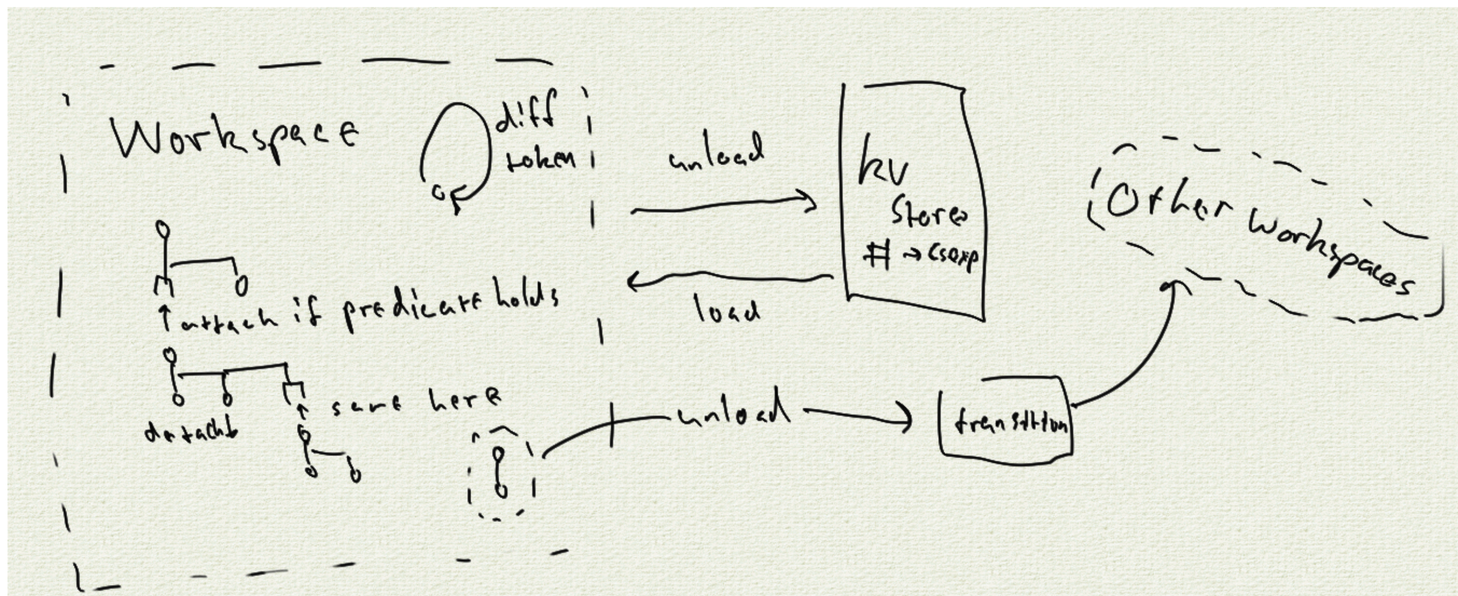
If we name this data `bigraph` then there are 6 slots under bigraph:

- A first slot is raw binary data encoding a `nxm` bit-matrix for P->T.
- Similarily the second slot is encoding a `mxn` bit-matrix for T->P.
- Then we have four lists; length m: places, length n: transitions, length equal to number of 1 bits in P->T: how to call predicates, length … T->P: how to call procedures.

Version Control

This data description gives us some kind of “denotational semantics” but from the operational standpoint it makes more sense to look at the places as version control systems (and the transitions as build systems).

This makes sense since the transitions are concerned with code while the places keep track of data. Our data is typed (via predicates) and represented as trees (csexps) so a recent paper applies cleanly (https://doi.org/10.1145/3453483.3454052 (https://doi.org/10.1145/3453483.3454052))



This picture is based on the ideas in the paper, using content addressing and by targeting a simple editscript we can compile a diff for (typed) ordered trees in linear time.

Each place (in the supply chain) can find shared structure and keep track of the content addresses it has seen. This makes it easier to program the places to be idempotent (which can grant us coordination-freeness if we satisfy propnet axioms) and allows us to version interfaces.

When we change what data can be associated with a name then we need to “create a new name” which is why (by convention) I expect every datalisp name to be indexed by a version: `(4:name1:0...)` i.e. `name` is version `0`. If the graph updates arrive via the graph then the version<->implementation relation should converge by removing confusion (paper linked below).

Both places and transitions will usually be equivalency classes since whenever any program is updated there is one more version to choose from. The goal of datalisp is to be a decentralized name system, i.e. to resolve that ambiguity and allow users to refer to some name, qualifier free, and nothing unexpected happens (i.e. “do what I mean”).

The idea is that the structure above will be grown and pruned in ways that preserve important invariants but this is mostly out of scope for this paper.

In order to reason about updates we will need a measure theory to decide whether or not a patch can be trusted. This is quite complicated, but the idea is to have `(signed (contextual (measured (named data))))`. The `measure` refers to the prior of the signee, then hopefully we can use something like: https://arxiv.org/pdf/1710.04570.pdf (https://arxiv.org/pdf/1710.04570.pdf) to improve the confidence we can have in our assessments.

Interaction

While creating templates is an important piece of the system there are some other fundamental interfaces:

- Tree editor :: Structured editor, similar to what you'd use to interact with a filesystem, used to make syntactically correct changes to csexps.
- Launcher :: List of human readable options with a minibuffer that shows extra data about highlighted option; such as the command/template to give to `bash` if option is selected (in case of template will give you a form to fill).
- Form filler :: Combines the two menus above with a "shell language" which translates back to canonical S-expressions and firing sequences in a bigraph of namespaces.

There are more editors required for example for the bigraphs and for the string diagrams but these are being worked on by https://statebox.org (https://statebox.org) and in the worst case could always be described as data using the general purpose tools described above.

The structured editor will anyway need to support predicate checking data in the tree and external viewers / editors (for images/videos/etc) so writing a simple GUI editor for creating the bit matrices should be low hanging fruit (attaching the correct data can then be done in the menu system).

The shell language is based on "generalized S-expressions" which not only allow writing `(f ...)` as `f(...)` but also borrow some syntax from nim to allow writing `f(a ...)` as `a.f(...)` this makes tab completion (a problem for datalog…) much more powerful (as is familiar to IDE user in OOP languages).

However it doesn't end there; since we don't care in which order the template is filled we can add a bit more syntactic sugar `(f a b c)` is `(f' b c a)` is `(f'' c a b)`. Finally we also have three types of parenthesis: `(place)`, `{predicate}` and `[procedure]`. These are different access patterns to the data in the bigraph and again it makes the tab completion more "complete".

The semantics of this metaprogramming language are just quotation and composition. This is pretty obvious when we consider the canonical form (each netstring is quoted and the first one names the template that says how to compose the rest) but in the generalized form we are whitespace sensitive so we need to have a way to "box" data (since we don't want to write prefix lengths by hand), in practice this will be a keybind or some mnemnomic like `k:` that lets you inside the box and then you can type whitespace without splitting the box.

For the braces and the brackets we are accessing programs directly so that we can run them or test them (rather than filling in a template and sending it to the transition you just invoke the transition directly with what you type in: `sh[cat myfile.txt]`).

What I am describing is a user interface to access a database (with stored procedures) so why do we want the canonical S-expressions as a representation? It doesn't seem very useful as a packet format nor as an on-disk format, but it is very useful as a canonical representation format for content addressing (~DHT) or proof-carrying data.

Coordination

What I've described so far is all from the perspective of a single peer organizing their operating system but I am also investigating how multiple peers can coordinate.

I haven't defined anything so far but the discussion surrounding these concepts is so confused that I will be a bit more rigorous now.

A distributed system is coordination-free if it doesn't matter in which order the information propagates around the system (the result is always the same).

Two peers are `coordinated` if they happen to take the same quotient of a coordination-free space (identify data in the same way - i.e. they can use abbreviations amongst themselves to compress their communication).

A coordinated subgraph is some part of your bigraph that is identical to the state agreed on by consensus (with someone).

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 (OS, ~automatic updates) while join consensus is a peer consenting to some configuration (install software).

- Meet consensus :: the peers have places seeding "pinned" data, which is kept stable for bootstrap (syncing is cheaper for everyone if you can bootstrap from the meet) all further development references the meet => it's a "release" of the software.
- Join consensus :: is when a peer decides to add customizations or patches from elsewhere to their graph. Many people in the same `join` can also form a `meet` , each `meet` is treated as an immutable checkpoint. We can measure legitimacy of a meet in (weighted) coordination.

Note that each peer is only able to make probabilistic estimates about the outside world so they may not actually form global consensus but the economic incentives (protocols) should be designed to encourage overlap.

Quotients are created by identifying equivalent data and choosing a canonical representative of the semantic. 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.

This is happening at "human speed" or "email speed" and "byzantine fault tolerance" (or "tragedy of the commons"-resistance) is not ensured algorithmically (maybe one day someone will find out how to do that - the system is designed to make it easier to find such a solution - but in the absense of a solution we must approximate).

Legitimacy is decided by the user but is influenced by the network (based on user settings).

Economics

Donald G. Saari has a book called Basic Geometry of Voting where he identifies that Arrows impossibility theorems are a result of ballots not having enough information to have a reliable tally method. Since then we've come up with a lot of interesting cryptographic constructions but so far no way to assemble them in such a way that we can avoid the tragedy of the commons in an open system.

The way I intend to get started performing experiments with the blueprint: `(signed (contextual (measured (named data))))` is to have:

- `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)` where the data we hash is `(1:2y:somez:data)`.
- `named` is the datalisp identifier for predicates, templates and procedures that the anonymous data will be subject to upon arrival. The data is valid datalisp since the ascii decimals only need that many slots of data they are the simplest "name" and a good way to move around groups of "wires" for our wiring/string diagram editors the names have more constraints that the incoming data will be tested with but they have a fixed number(&order) of slots for wires.
- `measured` means that the signee has voted with an assessment (which is implictly a bet) initially this will be of the form (Signal:Noise, Amount), a bet on certain odds.
- `contextual` refers to a way to tally the vote, the context that you are participating in coordinating will have some rules evaluating the measures and the participants should be able to coordinate their progress by following the protocol (and eventually maybe the protocol can be made byzantine fault tolerant, tragedy of the commons resistant and provably fair).

These metadata can be exchanged over IRC (as bootstrap) and the political systems can be used to govern the network (kicking / banning people) with open registration (send data conforming to some name to nickserv).

Currently the way I am factoring the process of "economics" into roughly:

- Voting :: Vote on whether a piece of data was signal or noise in the given context. In order to vote you must be eligible in some way that is not clear yet.
- Tallying :: Collect votes and use the predetermined method to tally the vote, produce proof of result. Now others can refer to this proof to resolve their bets, creating a kind of UTXO.
- Exchanging :: Influence can be exchanged (voting rights)
  - Burning :: Burn influence and earmark it for someone else (make space for someone else)
  - Minting :: Create influence and thereby reveal your relative assessment of relative worth of influence w.r.t. burner.

The idea is to try and make the exchange game and the vote game balance each other out. One way I thought to do that is to embed exchange transactions into the source tree that all the reproducible builds in the network depend on. That way the block reward will go to the developers who own the patch.

Implications

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 *cough*X*cough*).

Having canonical representations for some semantics like keybindings makes it easier to compile configurations for different choices of window manager / text editor / etc. Which then makes it easier for a new text editor / window manager to know what kind of interface they need to support to be painlessly compatible with the interface people actually use.

Similarily 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 (still reliant on sh).

Menus can be generated or fetched on the fly so you can crawl and construct this “internet index”, persist interesting information, open it in external viewers of some sort or edit/propagate it.

Ideally we want to be able to serialize any state, reproduce the program and restore the state we had serialized. Of course the menu system I am building to bootstrap the system will follow this principle but I would like to integrate a window manager (so you can save and restore workspaces) and eventually have a system usable by a child or non-technical user.

Datalisp is supposed to be very easy to implement, the syntax is as simple as can be and the core semantics (which are wip) should not be much more complicated than datalog (calculating transitive closures and queries). This is because I would like to be able to have simple devices that can rely on my other machines for most (back-end) functionality.

Where things get more complicated is on the machines that talk to the untrusted network, since keeping track of probabilities is probably going to get arbitrarily complicated once market incentives are in the mix. 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; HTML is notoriously permissive with syntax errors and in general it is pretty terrible in many ways. Especially for 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.

Furthermore I believe that lisp itself could 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 allow us to balance censorship and pollution and overcome the lisp curse.