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).

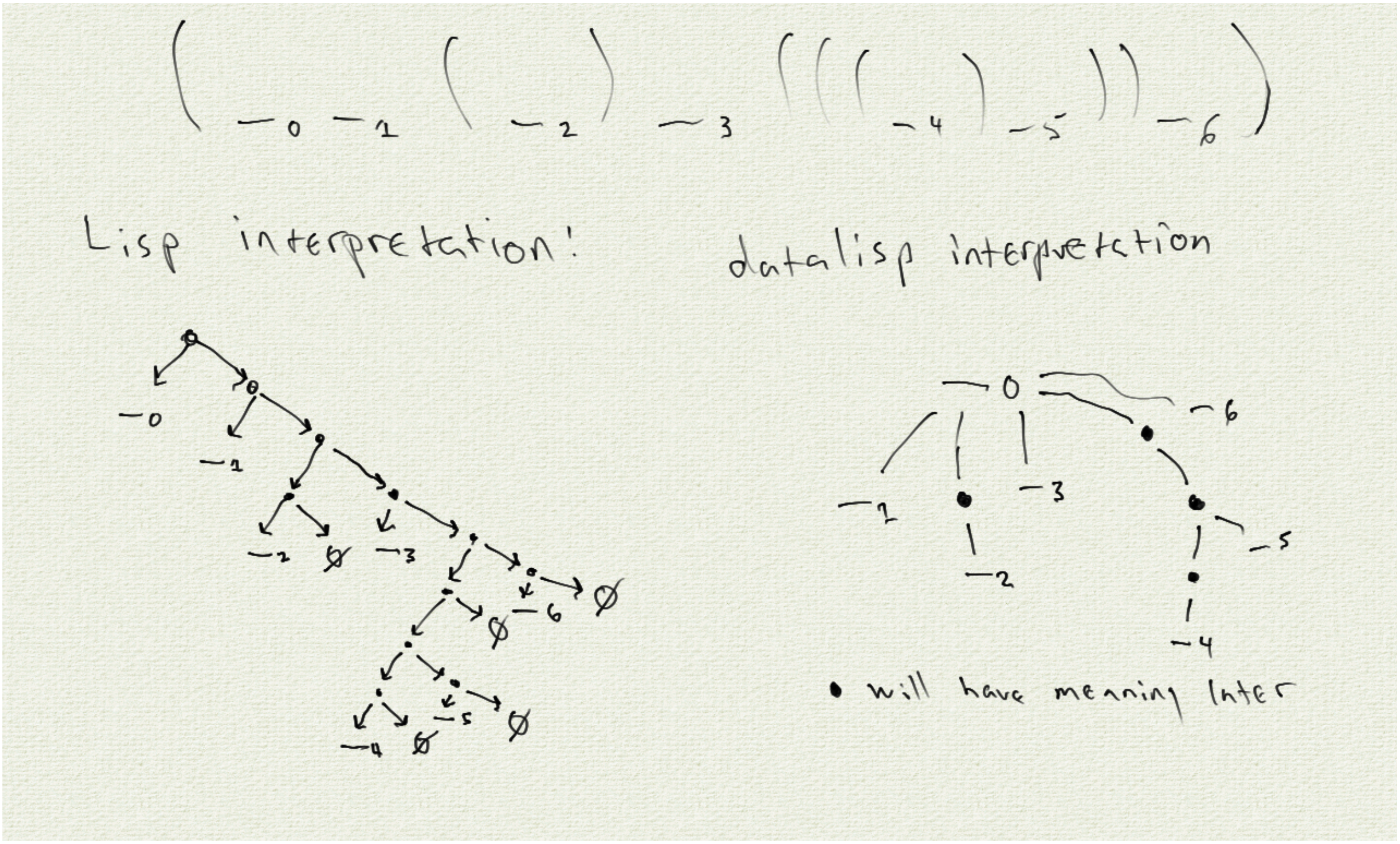
Introducing Datalisp

A Motzkin path is a word with digits from the alphabet $\Sigma = \{)\,(\,\_\}$ subject to the constraint that parenthesis must be balanced.

Canonical S-expressions 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 dependency lattice.

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



To understand the datalisp interpretation we must understand some of the higher level ideas.

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). 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.

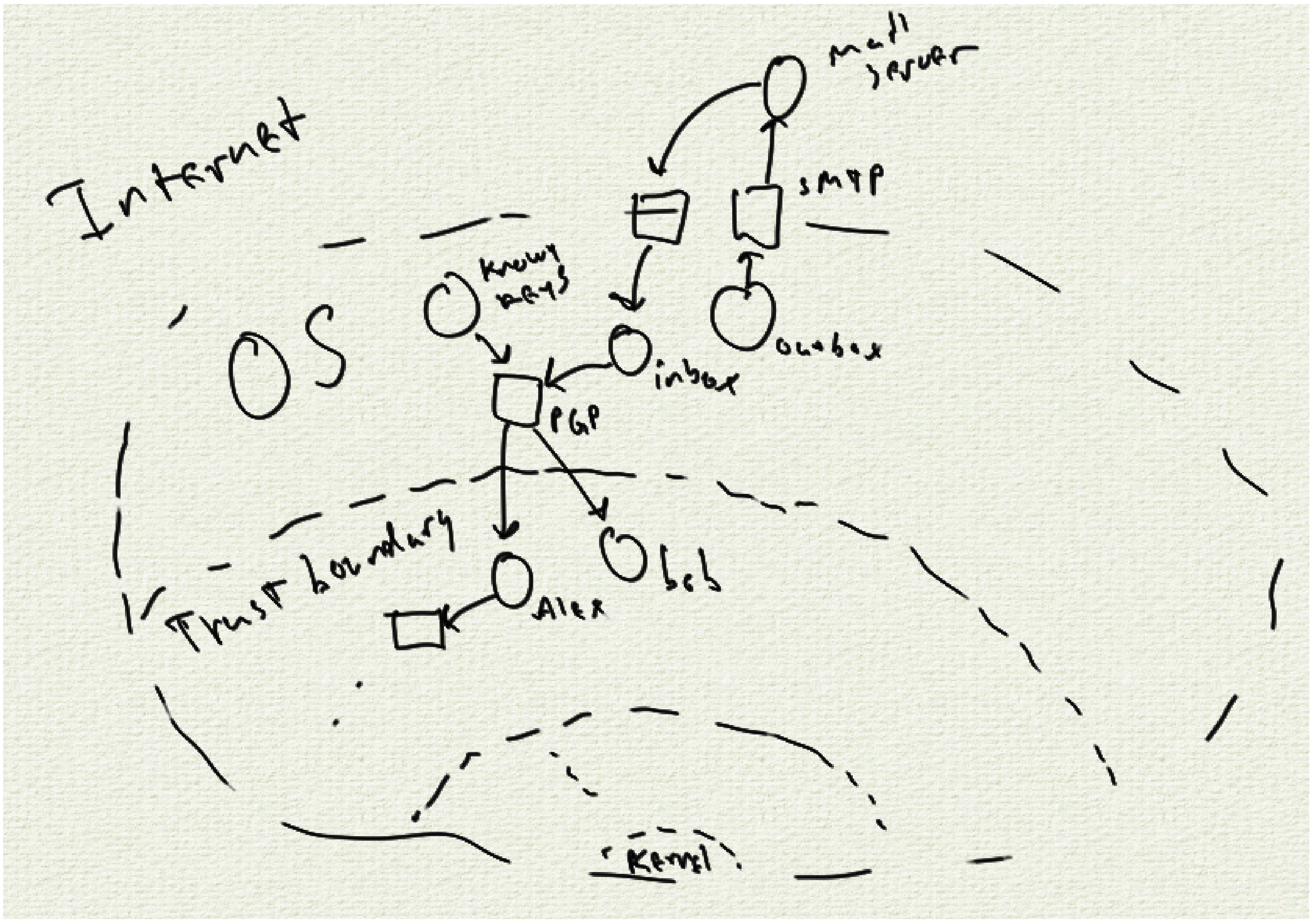
As a result we can have a joinsemilattice of templates where we have to fill in the fringe before proceeding up the structure; collapsing full templates into slots along the way till we've built the resulting file.

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.

Consider the following sketch:



There are potentially many ways to understand this sketch but we disect it (roughly, wip…) as follows:

- Squares are called “transitions” :: they are represented by a guix manifest (describing how to reproduce the programs needed to perform the transition).
- Circles are called “places” :: they are represented by a namespace, each name specifies how the associated data will be tested, the frames it will be injected into and which transitions the resulting data can be sent to.
- Circle->Square arrows are predicates i.e. the “how associated data will be tested”
- Square->Circle arrows are procedures i.e. the “frames to inject and transitions to send the data to” but also how the resulting data will be quoted and named (in the target places).

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.
- Similarly 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: predicates, length … T->P: procedures.

Version control